New Multiobjective Shortest Path Algorithms

Dissertation

zur Erlangung des Grades eines Doktors der Naturwissenschaften
(Dr. rer. nat.)
am Fachbereich Mathematik und Informatik
der Freien Universität Berlin

vorgelegt von

Pedro Maristany de las Casas

Berlin 2023

Erstgutachter: Prof. Dr. Ralf Borndörfer
Zweitgutachterin: Prof. Dr. Dorothea Wagner
Tag der Disputation: 15.05.2024

DECLARATION OF AUTHORSHIP

I declare to the Freie Universiät Berlin that I have completed the submitted dissertation independently and without the use of sources and aids other than those indicated. The present thesis is free of plagiarism. I have marked as such all statements that are taken literally or in content from other writings. This dissertation has not been submitted in the same or similar form in any previous doctoral procedure.

I agree to have my thesis examined by a plagiarism examination software.

Date: Signature:

ZUSAMMENFASSUNG

In dieser Arbeit beschäftigen wir uns hauptsächlich mit dem *Multikriterielle Kürzeste Wege* (MOSP) Problem. Es ist eine Verallgemeinerung des klassischen Kürzeste Wege Problems, bei der Kanten im Eingangsgraphen mit d-dimensionalen Vektoren anstelle von Skalaren gewichtet sind. Der Hauptbeitrag der Arbeit ist der *Multiobjective Dijkstra Algorithmus* (MDA), ein label-setting Algorithmus, der in der Theorie und in der Praxis eine Verbesserung der in der Literatur vorhandenen Ergebnissen darstellt. Motiviert durch dieses Ergebnis entwickeln wir im weiteren Verlauf der Arbeit Varianten des MDAs für verschiedene Varianten des MOSP Problems.

Für das Punkt-zu-Punkt MOSP Problem hat der *Targeted MDA* (T-MDA) die gleiche asymptotische Laufzeit wie der MDA. Durch einen erhöhten Speicherverbrauch ist er in der Praxis aber deutlich schneller. Die zusätzlich gespeicherten Pfade werden auf eine pseudo-lazy Art verwaltet. Dies ist eine neuartige Methode zur Organisation von explorierten Pfaden, die sicherstellt, dass die Größe der Prioritätswarteschlange des Algorithmus auf höchstens einen Pfad pro Knoten im Graphen beschränkt bleibt. Die Pfade, die aus der Warteschlange zurückgehalten werden, werden in Listen organisiert, die durch Voranstellen oder Anhängen von Pfaden sortiert bleiben. Das heißt, dass nur Konstantzeit-Operationen benötigt werden. Der T-MDA löst größere Instanzen als der MDA und ist auch schneller.

Wir untersuchen die Verallgemeinerung des zeitabhängigen Kürzeste Wege Problems auf den multikriteriellen Fall. Wir bieten eine detaillierte Analyse der Grenzen und Möglichkeiten dieser Verallgemeinerung und diskutieren, wann der MDA hierauf anwendbar ist. Für MOSP Instanzen mit großen Mengen optimaler Pfade sind gute Approximationsalgorithmen wichtig. Wir kombinieren den MDA mit einer Technik zur Partitionierung des Ergebnisraums aus der Literatur, um einen neuen FPTAS für das MOSP Problem zu erhalten. Der resultierende MD-FPTAS funktioniert auch für Instanzen mit verallgemeinert zeitabhängigen Kostenfunktionen, was neuartig ist.

Schließlich verwenden wir den MDA und seine bikriterielle Version als Unterroutinen, um jeweils das Multiobjective Minimum Spanning Tree (MO-MST) Problem und das k-Shortest Simple Path (k-SSP) Problem zu lösen. Eine MO-MST Instanz wird gelöst, indem der MDA auf einen sogenannten Übergangsgraphen angewendet wird. In diesem Graphen haben Pfade äquivalente Kosten zu den Bäumen im ursprünglichen Graphen und somit entsprechen die optimalen Lösungen, die vom MDA im Übergangsgraphen berechnet werden, den optimalen Bäumen im ursprünglichen Graphen. Da der Übergangsgraph eine exponentielle Größe im Verhältnis zur Größe des ursprünglichen Graphen hat, diskutieren wir neue Techniken zur Reduzierung seiner Anzahl von Kanten. Die Lösung des k-SSP Problems, ein skalares Optimierungsproblem, unter Verwendung einer bikriteriellen Unterroutine ist überraschend, aber unser neuer Algorithmus ist sowohl in der Theorie als auch in der Praxis auf dem neuesten Stand.

ABSTRACT

The main problem studied in this thesis is the *Multiobjective Shortest Path* (MOSP) problem. We focus on the problem variant with three or more objectives. It is a generalization of the classical Shortest Path problem in which arcs in the input graph are weighted with vectors instead of scalars. The main contribution is the *Multiobjective Dijkstra Algorithm* (MDA), a label-setting algorithm that achieves state of the art performance in theory and in practice. Motivated by this result, the thesis goes on with the design of variants of the MDA for different variants of the MOSP problem.

For the One-to-One MOSP problem the *Targeted MDA* (T-MDA) has the same asymptotic running time than the MDA but trades in memory for speed in practice. The additional paths stored during the T-MDA are managed in a pseudo-lazy way. This is a novel way to organize explored paths that ensures that the algorithm's priority queue stores at most one path per node in the input graph simultaneously. The paths that are held back from the queue are organized in lists that are kept sorted by just prepending or appending paths to them, i.e., using constant time insertions. The resulting implementation of the T-MDA solves bigger instances than the MDA and is also faster.

We study the generalization of the Time-Dependent Shortest Path problem to the multiobjective case. We provide a detailed analysis of the generalization's limitations and discuss when the MDA is applicable. For MOSP instances with large sets of optimal paths, good approximation algorithms are important. We combine the MDA with an outcome space partition technique from the literature to obtain a new FPTAS for the MOSP problem. The resulting MD-FPTAS works also for multiobjective instances of the Time-Dependent Shortest Path problem, which is a novelty.

Finally, we use the MDA and its biobjective version, the BDA, to solve the Multiobjective Minimum Spanning Tree (MO-MST) problem and the k-Shortest Simple Path (k-SSP) problem, respectively. For the solution of instances of these two problems, the MDA and the BDA are used as subroutines. An MO-MST instance is solved applying the MDA on a so called transition graph. In this graph paths have equivalent costs to trees in the original graph and thus, the optimal solutions computed by the MDA in the transition graph correspond to optimal trees in the original graph. Since the transition graph has an exponential size w.r.t. the size of the original graph, we discuss new pruning techniques to reduce its number of arcs effectively. The solution of the k-SSP, which is a scalar optimization problem, using a biobjective subroutine is surprising but our new algorithm is state of the art in theory and in practice.

ACKNOWLEDGMENTS

During my PhD I have been enrolled in the project *Flight Trajectory Optimization on Airway Networks*, a cooperation with our partner Lufthansa Systems GmbH. Within the *Research Campus MODAL* it is funded by the *Bundesministerium für Bildung und Forschung*. I am fully aware of the extraordinary opportunity this role has been and I express my sincere gratitude for it.

My advisor, Professor Borndörfer, provided me with this opportunity. I have always enjoyed his trust in me and my skills, giving me the necessary room to develop as a researcher. It is that trust precisely that enabled me to find the right topic for this dissertation, one which, fortunately, kept me motivated. Trust, time, and guidance. Invaluable. Thank you, Ralf!

Along the journey I got to know Professor Sedeño Noda who, coincidentally, works in the university of my home town. His expertise in Multiobjective Combinatorial Optimization and his willingness to collaborate with us paved the road that led to this thesis. I will remember the lengthy hours we spent delving into problems during my visits to Tenerife as well as the undrinkable coffee in the Maths building cafeteria.

Professor Wagner agreed to be the second corrector of the thesis. The required time and effort in this role is considerable. I am very thankful that she accepted to invest them.

My colleagues and other researchers that I have met in the past years have been an enrichment in so many ways. Notable mentions go to Adam, Marco, Niels, and *l@s chic@s de Loli*, you have been a great inspiration and even greater friends.

My mother has always shaped my perception of effort and dedication. Her lessons and unwavering support kept me going also in discouraging times in a way that felt surprisingly natural. Her whole life she taught me, time and time again: at the end, it will be worth it. My brothers Eduardo and Pablo, Agustín, and my grandmother are my additional support to stay resilient when plans deviate from expectations. Thank you for your patience!

Finally, moving from Spain to Germany being 18 years old does, in my opinion, only work out if one manages to build a network of great friends. For my 30th birthday you woke up at 3am and came with me on a 15h hike from sea level to the top of the Teide. Many were not there but wanted to; you are all amazing! Rest assured, I will find a better place than this sheet of paper to express my gratitude.

The thesis was typeset using the ArsClassica LaTeX package by Lorenzo Pantieri.

PREFACE

This thesis puts together the results published in five papers: Maristany de las Casas, Sedeño-Noda, and Borndörfer (2021), Maristany de las Casas, Borndörfer, et al. (2021), Maristany de las Casas, Kraus, et al. (2023), Maristany de las Casas, Sedeño-Noda, and Borndörfer (2023), and Maristany de las Casas, Sedeño-Noda, Borndörfer, and Huneshagen (2023). My aim has been to improve the exposition that I would have gotten by merging these five papers together in a cumulative dissertation. To establish comparability when it is needed I also re-ran multiple experiments using newer versions of the algorithms' implementations and partially also different instances. Thus, results presented here are not comparable to the ones in the original publications. The bottom lines in the papers and in the thesis coincide.

All implementations used throughout the thesis with exception of the one closely related to the code used by our industry partner Lufthansa Systems GmbH (Chapter 11) are publicly available in my GitHub. Additionally, the repositories contain the detailed results obtained from every instance, the scripts used to evaluate them, and the plots not included explicitly in the thesis. While it has gained more attention in the last five years, research on Multiobjective Shortest Paths was scarce prior to this period. A consequence was that most implementations were not publicly available. For this reason I re-implemented every benchmark algorithm in this thesis[1] s.t. the speedups possibly result in an isolated head-to-head comparison of the relevant techniques in every chapter. By doing so, the speedups reported in the thesis are sometimes lower than the ones reported in the original publications. I see it as a small price to pay to hopefully increase the relevance of the thesis.

I expect the thesis to be self contained regarding the used notions from Multiobjective Optimization. However, I point the reader to the book by Ehrgott (2005) for an in depth introduction to the field. A good introduction to Network Flow problems is the book by Ahuja et al. (1993). Necessary background on data structures and sorting techniques can be read for example in the book by Mehlhorn (1984). I use the Bachmann-Landau notation to describe the asymptotic behavior of algorithms throughout the thesis. An introduction is given in (Korte & Vygen, 2005, Section 1.2) and in the references therein.

I am the first author of all publications mentioned at the beginning of this preface. During my time as a PhD student I was the informal advisor (the formal advisor being Professor Borndörfer) of multiple Student Assistants and two of them wrote a Bachelor's or Master's thesis on the topic of two of the papers. Luitgard Kraus' Bachelor's thesis (Kraus, 2021) is on the content of (Maristany de las Casas, Borndörfer, et al., 2021) and Max Huneshagen's Master's thesis (Huneshagen, 2023) is related to the contents of (Maristany de las Casas, Sedeño-Noda, Borndörfer, & Huneshagen, 2023).

1 With exception of the KM algorithm used in Chapter 14.

CONTENTS

Part I

INTRODUCTION

1 MOTIVATION AND OUTLINE

It is difficult to trace back the history of the shortest path problem. One can imagine that even in very primitive (even animal) societies, finding short paths (for instance, to food) is essential. Compared with other combinatorial optimization problems, like shortest spanning tree, assignment and transportation, the mathematical research in the shortest path problem started relatively late.

(Schrijver (2012))

Even if mathematical research in the field started late, it has been a big success story in the past 60 years. The perceptibility of solution strategies for shortest path problems and their applicability in practice constitute a loop in which researchers keep adding features (resource constraints, time dependency, ...) to the original Shortest Path problem while still controlling the theoretical problem complexity and the computational running times. Each cycle in this loop brings shortest path problems closer to real world demands from industry or from everyday users (Geisberger et al., 2008; Delling et al., 2009; Delling & Wagner, 2009; Nannicini, 2009; Bast et al., 2015; Blanco et al., 2016; Blanco et al., 2017; Blanco et al., 2022; Baum et al., 2020; Euler et al., 2022). One of these problem variants that has gotten less attention so far is the Multiobjective Shortest Path (MOSP) problem in which *optimal* paths w.r.t. to multiple and conflicting optimization criteria are computed simultaneously.

1.1 THE APPLICATION BEHIND THE THESIS

The Network Optimization department at ZIB has been successfully cooperating for many years with Lufthansa Systems GmbH, a market leader in flight planning systems. As part of this industry cooperation that is funded by the German Federal Ministry for Research and Education (BMBF) within the context of the MobilityLab of the Research Campus MODAL, we could identify room for improvement in state of the art MOSP algorithms. Two of Lufthansa Systems's main cost drivers when planning aircraft trajectories are the flight duration and the fuel consumption. Often, an optimal solution for a single criterion scenario, using a weighted sum of time and fuel costs, is enough. However, whenever infrastructures become overstrained and big delays become a threat, a *front* of optimal paths w.r.t. both cost drivers gives decision makers in the planning stages after the optimization the possibility to choose a faster route even if more fuel is consumed. While doing so, choosing only between the fastest and the cheapest route, two single-criterion optimizations, might be to aggressive. This is how Multiobjective Optimization and its multiple solutions in the output fronts becomes an interesting modeling choice.

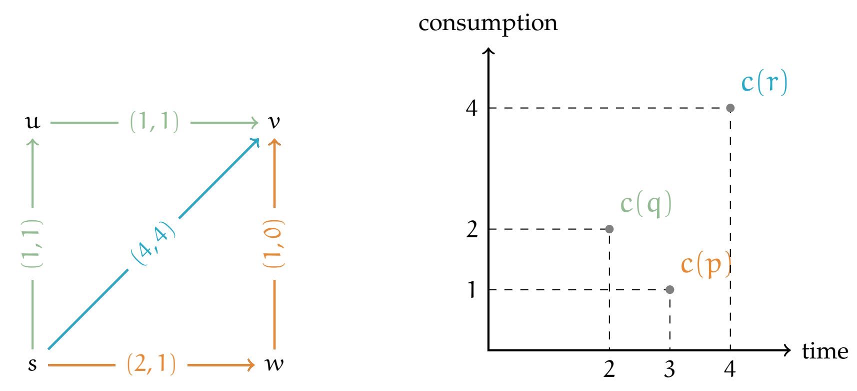
Lufthansa Systems was interested not only in MOSP instances with two cost criteria but wanted to assess the possibilities to model problems with three and more objectives. We started our journey delving into the MOSP literature with the aim of customizing a MOSP algorithm to suit Lufthansa System's requirements. During our search, we observed a scarcity of recent publications and noted that the fundamental methodology has remained unchanged since the introduction of E. Q. V. Martins (1984) classical label-setting MOSP algorithm.

Back then, in 2019, the publication of the Biobjective Dijkstra Algorithm (BDA) in (Sedeño-Noda & Colebrook, 2019) was very recent and we got in touch with the main author of the publication. A fruitful cooperation started. It led to the contributions/publications that are contained in this thesis. They are mostly detached from the Flight Planning application even though the gained knowledge is now being transferred back to Lufthansa Systems to be built in in their optimization core: one more cycle in the loop described in the first paragraph.

1.2 MULTIOBJECTIVE SHORTEST PATHS

An instance of the One-to-All MOSP problem consists of a weighted digraph $G = (V, A)$ and a source node $s \in V$. The weights defined on the arcs are d-dimensional vectors, $d \in \mathbb{N}$. For an arc $a \in A$ we refer to its weight as a's cost vector or simply a's costs. We assume throughout the thesis that the arc cost vectors only contain nonnegative components. Given a path in G, the cost vector of the path is the d-dimensional vector obtained after summing up the cost vectors of the arcs along the path. An s-v-path p for any $v \in V$ is said to be optimal if there is no other s-v-path q in G s.t. q is better than or equal to p in every cost dimension and strictly better in at least one dimension. This notion of optimality is commonly called *efficiency* in the Multiobjective Optimization jargon. If a path is not efficient, it is called a *dominated* path. For any node v the set of efficient s-v-paths in G is often called the *Pareto front* of s-v-paths in the literature. In this thesis, we simply call it the *set of efficient s-v-paths*. The goal in a MOSP instance is to find the set of efficient s-v-paths for every $v \in V$. The One-to-One problem variant includes a target node $t \in V$ in an instance's input and the goal is then to find the set of efficient s-t-paths. The problems are defined formally in Definition 3.2.

Example 1.1. Consider the example in Figure 1 and in particular the three s-v-paths in the graph on the left hand side . If we consider the orange path p and the green path q that consume 1kg and 2kg fuel, respectively, it is clear that p is optimal if consumption is the minimization criterion. However if we optimize also the paths' duration, things become more difficult. Traversing p takes 3min and traversing q takes 2min. The costs of p and q are $(3\text{min}, 1\text{kg})$ and $(2\text{min}, 2\text{kg})$, respectively. Optimizing both criteria simultaneously, it is not possible to decide which one is preferable. In fact, they are both efficient



(a) Subgraph with bidimensional arc costs. The first cost component represents the time needed to traverse each arc. The second cost component models the fuel consumption of an aircraft during the traversal of the arc. The paths p, q, and r are depicted in orange, green, and blue, respectively.

(b) Cost vectors of the paths on the left hand side on the plane. The paths q and p are efficient paths since their costs are nondominated. Both paths dominate the path r.

Figure 1: Exemplary Biobjective Shortest Path instance.

paths in the shown example. The blue path r however with costs $(4\text{min}, 4\text{kg})$ is a dominated path.

In Example 1.1 if r's cost vector would be $(1\text{min}, 4\text{kg})$, r would also be an efficient path. Hence, the structure of the arcs' costs influences the cardinality of the solution sets in MOSP instances. In fact, Hansen (1980) proved that there exists a directed graph with bidimensional arc costs and an exponential number of paths w.r.t. the graph's size in which all paths are efficient.

Going back to the practitioners' point of view, this opens up a dilemma. Single-criterion Shortest Path problems can be solved very fast but they deliver little flexibility in later planning stages. In contrast, a MOSP instance with arcs weighted using vectors can contain multiple interesting solutions but also too many to be either computed in a reasonable amount of time or to be practical.

1.2.1 Intractability

An optimization problem is said to be intractable if there is no algorithm that can solve all problem instances in polynomial time with respect to the size of the input (cf. Garey & Johnson, 1990, Section 1.3). A possible reason for a problem's intractability is that *"the solution itself is required to be so extensive that it cannot be described with an expression having length bounded by a polynomial function on the input length"* (Garey & Johnson, 1990, Section 1.4).

Even easy combinatorial optimization problems become intractable when their multiobjective siblings are considered. The reason is that the number of feasible solutions is usually exponential in the input size and cost vectors can be constructed s.t. every feasible solution is efficient. This is also the case for MOSP as proven by Hansen (1980). His results hold even for bidimensional arc costs. We use his MOSP instances throughout the thesis in our experiments. Given that the number of paths in a graph is exponential in the graph's encoding size, we end up with an exponentially sized output that a MOSP algorithm needs to compute, store, and output.

The intractability of discrete multiobjective problems does not hold researchers off from developing exact algorithms for these problems. In the case of MOSP, conventional wisdom in the Operations Research community states that the practical advantages of multiobjective models and multiobjective optimization outweigh the theoretical hurdles. In other words: *Even if the MOSP problem is hard in theory, it can be solved fast in practice.*

Instances of the classical Shortest Path problem on road networks are nowadays solvable in fractions of a second (cf. Delling et al., 2009; Bast et al., 2015) and the latest improvements focus on how to use ground breaking preprocessing techniques like e.g., Contraction Hierarchies (Geisberger et al., 2008) to reduce the number of iterations performed in the actual Shortest Path query. For MOSP, the situation was different when the research journey that led to the contents of this thesis started: the single iterations in state of the art algorithms contained inefficiencies with negative impact on both the algorithms' asymptotic running time bounds and their performance in practice. Thus, our goal was to design new MOSP algorithms that circumvent this inefficiencies and hopefully equip the research community with tools to further study MOSP applications knowing that fast algorithms are used under the hood. Thereby, we need to differentiate between instances with bidimensional arc costs and instances with higher-dimensional arc costs.

1.2.2 Biobjective Shortest Paths

A special case of MOSP is the *Biobjective Shortest Path* (BOSP) problem in which arc costs are two-dimensional vectors. Hansen (1980) published probably the first in depth study of the problem. For this special case, the research community has been more active than for general MOSP problems. The publication by Raith and Ehrgott (2009) gives a good overview of the algorithms used until 2009 to solve BOSP problems. Besides Martins' label-setting algorithm, *two phase* approaches were often used. Algorithms of this type search *supported solutions* in the first phase. Supported solutions are obtained by using weighted sums of the objective functions and solving the resulting single-criterion Shortest Path instances. Different methods from *Branch and Bound* to *Ranking* algorithms can be used to compute unsupported solutions in the second phase.

In Sanders and Mandow (2013) the authors discuss a parallel BOSP algorithm. The parallelized component is the algorithm's queue. They achieve an asymptotic running time improvement in their work. The work has two main limitations. First, the parallel queue is *"too complicated to be practical"* and is thus not implemented. Second, it is not known how to generalize the queue to MOSP instances with more than two arc cost components.

Two phase approaches for BOSP problems seem to be less relevant ever since, in 2015, Duque et al. (2015) published the *Pulse* algorithm. It was a novel approach since it is an exact BOSP algorithm that generates efficient paths by repeatedly calling Depth-First searches. The Pulse algorithm remained state of the art for some time until Sedeño-Noda and Colebrook (2019) introduced the first version of their *Biobjective Dijkstra Algorithm* (BDA). Since then, the BOSP community has had good news. One one hand, the publication of the BDA improved the best known asymptotic running time bound for BOSP and raised the bar regarding the solvability of large scale BOSP instances in a reasonable amount of time. On the other hand, some groups from the Artificial Intelligence community started to be interested in the topic. Ulloa et al. (2020) published a Biobjective A^* (BOA^*) algorithm for One-to-One BOSP. The most notable work on the topic was however published one year later by Ahmadi et al. (2021). Their Enhanced Biobjective A^* ($\text{BOA}^*_{\text{enh}}$) algorithm is a milestone for BOSP solvability. In our first version of the preprint (Maristany de las Casas, Kraus, et al., 2021) we enhanced the BDA with the techniques used in (Ahmadi et al., 2021) and achieved better running times. However, the authors informed us that they had improved their (publicly available) implementation of the $\text{BOA}^*_{\text{enh}}$ algorithm and were faster than the improved BDA. Since this version of the BDA pairs the results of two publications without further contribution and is currently slower than the $\text{BOA}^*_{\text{enh}}$ algorithm, we decided not to include it in this thesis. However, an in-depth comparison of the asymptotic running time of the BDA with other BOSP algorithms is not included in Sedeño-Noda and Colebrook (2019). We include it as a byproduct-contribution in this thesis.

1.2.3 Higher Dimensional Multiobjective Shortest Paths

After the biobjective case was studied by Hansen (1980), the generalization to higher-dimensional arc cost vectors started to be considered in the literature. White (1982) and Loui (1983) considered first variants of the Multiobjective Shortest Path problem. In particular Loui (1983) already recognizes the Dynamic Programming structure of the problem and gives an exact algorithm. He works in a maximization and thus, as in the Longest Path problem, heavy on assumptions to guarantee that Dynamic Programming is applicable.

Similar to Dijkstra's algorithm (Dijkstra, 1959) for the Shortest Path problem, the MOSP algorithm by Martins (E. Q. V. Martins, 1984) is a *label-setting* algorithm (cf. Section 3.4) and remained state of the art for decades after it was published. The exposition in the original paper is coined to cover One-to-One MOSP problems but it is clear from how the problem is stated and how the algorithm (E. Q. V. Martins, 1984, Algorithm 1) is designed that it is also a One-to-All MOSP algorithm. We give an overview of the literature for One-to-One MOSP problems in Chapter 9.

First Guerriero and Musmanno (2001) and later Paixão and Santos (2013) perform extensive computational analysis of different labeling and sorting methods in the design of One-to-All MOSP algorithms. On different artificial graphs, they conclude that *label-correcting* methods outperform *label-setting* methods. Making use of the *dimensionality reduction* technique first used in (Pulido et al., 2015) in the context of MOSP problems, the superiority of label-setting algorithms seems nowadays granted. Dimensionality reduction (cf. Section 3.5) is used in conjunction with lexicographic sorting of paths according to their costs and with a priority queue to design today's state of the art MOSP algorithms.

An interesting concept to compare the asymptotic behavior of MOSP algorithms is the notion of *output sensitivity*. An output sensitive asymptotic running time bound for an algorithm is a bound that depends on the algorithm's output size and possibly also on its input size. Throughout the thesis, we are interested in output sensitive running time and memory consumption bounds that are polynomial in the size of the input and the output. If an algorithm achieves such a bound for every instance of the problem it solves, we say that it is an *output sensitive algorithm*. Martins does not bound the running time of his MOSP algorithm in (E. Q. V. Martins, 1984). The lack of details regarding data structures makes it hard to derive a bound from his pseudocode. However, it is well known that the One-to-All MOSP problem is solvable using output sensitive algorithms (see e.g., Bökler, 2018). Interestingly, Bökler (2018) also proves that this does not hold for the One-to-One MOSP problem unless $P = NP$.

Consider a One-to-All MOSP instance with d-dimensional arc costs $d \geqslant 3$. We denote the number of nodes and the number of arcs in the input graph by n and m, respectively. Moreover, we denote the overall number of efficient paths by N and the maximal number of efficient s-v-paths for a node $v \in V$, by $N_{\max}$. Demeyer et al. (2013) published an improved version of Martins's algorithm. As described in their paper, the algorithm has an output sensitive running time bound of

$$\mathcal{O}\left(N \log(dnN_{\max}) + dmnN_{\max}^2\right). \tag{1}$$

In (Breugem et al., 2017) the authors also use Martins's algorithm for their contribution. In Lemma 3.1. of their paper, they derive an output sensitive running time bound of $\mathcal{O}\left(nN^2\right)$ for Martins's algorithm, which is worse than (1). In Chapter 5 of this thesis we discuss a version of Martins's algorithm that has a running time bound of

$$\mathcal{O}\left(N \log(dn) + dmnN_{\max}^2\right). \tag{2}$$

While at a first glance it is debatable to what extent our version still adheres to the original Martins's algorithm, we believe that we are only interpreting the high level pseudocode in (E. Q. V. Martins, 1984, Algorithm 1) in the best possible way we can think of to make our benchmarks as meaningful as possible.

The Multiobjective Dijkstra Algorithm (MDA) is the generalization of the BDA to general-dimensional arc costs and is, together with its variants, the main contribution from the thesis. Using the same notation as before, the MDA has an output sensitive running time bound of

$$\mathcal{O}\left(N \log(dn) + dmN_{\max}^2\right). \tag{3}$$

How do these output sensitive bounds translate into practice and into the mentioned *conventional wisdom* of MOSP being hard in theory by solvable in practice? Finding solid answers to these questions is our main goal in Part ii and Part iii of the thesis.

1.2.4 The Algorithm's Name

Besides being label-setting algorithms for MOSP it is not clear at the first glance why the name for the *Biobjective Dijkstra Algorithm* (BDA) and the *Multiobjective Dijkstra Algorithm* (MDA) is justified. Time will decide whether the current naming of the BDA and the MDA is kept. An anecdote to end the introduction:

Möhring (1999) describes Martins's MOSP algorithm for instances with bidimensional arc costs. He states, loosely translated from German:

> For every efficient path, the algorithm is as fast as Dijkstra's algorithm.
>
> (Möhring (1999, Section 2.1, Page 9))

Since today's implementations of Dijkstra's algorithm are extremely fast, we can interpret these words as the possibility to solve MOSP instances if the number of efficient paths remains small. This is indeed the case when, for example, shortest paths on road networks are searched w.r.t. time and w.r.t. distance. Both objectives tend to be strongly correlated and thus, not many efficient paths exist between two nodes.

However, Dijkstra's algorithm runs in $\mathcal{O}\left(n\log(n)+m\right)$ time. Thus, none of the above bounds for Martins's algorithm second Möhring's claim. Since he is this thesis's author first optimization lecturer, we want him to be right. Indeed, using the BDA, his statement holds. The running time (3) of the MDA drops to $\mathcal{O}\left(N\log n+mN_{\max}\right)$ in the biobjective case (cf. Sedeño-Noda & Colebrook, 2019), i.e., for the BDA. By definition, we have that $N \leqslant nN_{\max}$ and thus, we can write the last bound as a worse bound and obtain the output sensitive running time bound

$$\mathcal{O}\left(N_{\max}(n\log(n)+m)\right).$$

The term states that the running time of the BDA is bounded by the size of the largest set of efficient paths multiplied by the running time of Dijkstra's algorithm. This is probably a better reason than just being a label-setting BOSP algorithm for the naming choice leading to the BDA's name. Since the MDA is a generalization of the BDA to the general-dimensional case and its running time bound (3) is lower than the bound (2) for Martins's algorithm, its name feels justified.

1.3 CONTRIBUTIONS AND OUTLINE

The thesis has three parts besides this introduction. Chapters not mentioned in this outline are introductions or conclusions.

In Part ii we discuss label-setting algorithms for MOSP. In Chapter 4 we introduce the MDA. This is the main contribution in this thesis. Immediately after, in Chapter 5, we discuss our new version of Martins's algorithm. This chapter is needed to better understand the relevance of the MDA and also to get a deeper insight into the results obtained in Chapter 6 in which we benchmark both label-setting MOSP algorithms against each other. For our benchmarks we use large scale three-dimensional instances and manage to solve instances that were unsolved so far in the literature. We also add some four-dimensional instances that are defined on artificial graphs.

Part iii is devoted to making the MDA more suitable for practical purposes. First, in Chapter 9, we design a One-to-One version of the MDA. The resulting version is called the *Targeted Multiobjective Dijkstra Algorithm* (T-MDA). It trades in a higher memory consumption than the MDA to speed up its single iterations. Moreover, using A^*-like techniques, it reduces the number of iterations needed to solve a given One-to-One MOSP instance. In Chapter 10 we study the possibilities to generalize the MOSP problem to a setting in which arc costs depend on the costs with which the tail node of the arc is reached. In other words, we study possible generalizations of the Time-Dependent Shortest Path problem to a multiobjective scenario. We call it a *dynamic costs* setting. While in this chapter we end up with a setting in which the MDA and the T-MDA work out of the box, discussing the modeling possibilities in the studied scenario is, to the best of our knowledge, an addition to the available literature on the topic. Part iii ends with the design of a new *FPTAS* for MOSP based on the MDA. We describe the new approximation algorithm in Chapter 11. It combines the MDA with a partition of the output set of costs from the literature. The partition bounds the output size with a term that is polynomial in the input size and $1/\varepsilon$ for any given value of $\varepsilon > 0$. We then use the resulting *MD-FPTAS* to solve instances of the *Horizontal Flight Planning* problem. Instances of the problem have multiobjective dynamic arc cost functions and the output sets in the exact scenario are too dense for practical purposes. Thus, the study of the performance of the MD-FPTAS in this scenario is particularly interesting.

Finally, in Part iv we discuss two problems in which the MDA can be used as a subroutine. In Chapter 13 we solve the *Multiobjective Minimum Spanning Tree* problem using a version of the MDA that handles the search graph implicitly. We see this chapter as an example of how the MDA can be used to solve discrete Multiobjective Dynamic Programming problems. The last contribution in the thesis is discussed in Chapter 14 where we study the k-Shortest Simple Path problem. Using a recent black box algorithm from the literature, it can be solved by solving $\mathcal{O}(k)$ instances of the *Second-Shortest Simple Path* problem. We show that this subroutine can be solved efficiently using a modified version of the BDA.

The introduction to the thesis so far serves in particular as an introduction for Part ii. The other two parts have their own introduction and conclusion as they are unrelated and mirror different stations of our MOSP-journey.

2 NOTATION: MULTIOBJECTIVE DISCRETE OPTIMIZATION AND GRAPHS

In this chapter, we briefly introduce notions from the field of *Discrete Multiobjective Optimization* (DMO) that are needed throughout the thesis. For an in depth discussion of the topic, we refer the reader to Ehrgott (2005) and Emmerich and Deutz (2018). We also settle our notation for directed graphs and paths therein. Formal definitions can be found in (Ahuja et al., 1993).

2.1 DISCRETE MULTIOBJECTIVE OPTIMIZATION

An instance of a d-dimensional DMO problem, $d \in \mathbb{N}$ considers a discrete set X of feasible solutions and d cost functions $c_i : X \to \mathbb{R}$, $i \in \{1, \dots, d\}$, map every $x \in X$ to its unique cost vector $c(x) := (c_1(x), \dots c_d(x)) \in \mathbb{R}^d$. d is the number of objectives of the instance at hand, also referred to as the dimension of the instance. The set X is called the *state space* and the set $Y := \{c(x) \mid x \in X\}$ is called the *outcome space*. For a subset X' of feasible solutions, we denote their cost vectors by $c(X') := \{c(x') \mid x' \in X\}$. Optimality is defined using the following strict partial order on Y.

Definition 2.1 (Dominance relation)**.** Consider a subset $Y \subseteq \mathbb{R}^d$ for some $d \in \mathbb{N}$. Let $y, y' \in Y$ be two vectors. y *dominates* y' if and only if $y_i \leqslant y'_i$ for all $i \in \{1, \dots, d\}$ and $y \neq y'$. If x dominates y, we write $x \prec y$.

Then, in Multiobjective Optimization optimality is defined as follows.

Definition 2.2 (Nondominance and Efficiency)**.** Using the notation from this section, consider a tuple $(X, c, Y, \prec)$ encoding a d-dimensional DMO instance.

- Let x, x' such that $c(x) \prec c(x')$. We say that x *dominates* x' and $c(x)$ *dominates* $c(x')$.
- $y \in Y$ is called *nondominated* if and only if there is no $y' \in Y$ s.t. $y' \prec y$.
- $x \in X$ is called an *efficient solution* if $c(x) \in Y$ is nondominated.
- $Y^* := \{y \in Y \mid y \text{ is nondominated}\}$ is called the *nondominated set*.
- $X^*_{\max} := \{x \in X \mid x \text{ is efficient}\}$ is called the instance's *set of efficient solutions* or *maximum complete set of efficient solutions*.
- $X^*_{\min} \subset X^*_{\max}$ containing exactly one efficient solution for every nondominated vector is called a *minimal complete set of efficient solutions*.

The nondominated set and the maximum complete set of efficient solutions of a DMO instance are unique. However, like it is often the case in single-criterion optimization, we are not interested in the set of all efficient/optimal solutions. Instead, it suffices to have one efficient solution for every nondominated cost vector. Throughout this thesis, we thus discuss algorithms that output a minimal complete set of efficient solutions. For this reason, we drop the min subscript when referring to minimal complete sets of efficient solutions.

Definition 2.3 (Dominance or Equivalence Relation)**.** For solutions x, $x' \in X$ we write $x \preceq x'$ if $c(x) \leqslant c(x')$. For a set $X' \subseteq X$ and an $x \in X$ we write $X' \preceq_D x$ or $c(X') \preceq_D c(x)$ if there is $x' \in X'$ s.t. $x' \preceq x$.

The $\preceq_D$ operator is useful to build a minimal complete set of solutions iteratively. If we have a set X' containing only efficient solutions with pairwise different cost vectors and we consider a candidate solution $x \notin X'$, then $X' \preceq_D x$ holds if X' contains a solution that dominates x or a solution that is cost equivalent to x.

We assume the reader is familiar with the asymptotic analysis of algorithms. An introduction can be read in (e.g., Garey & Johnson, 1990; Cormen, 2022). Let $|X'|$ denote the cardinality of X'. Then the following statement holds.

Proposition 2.1 (Complexity of dominance checks)**.** *Let X' be a set of feasible solutions of a d-dimensional DMO instance. For a feasible solution x', the check $c(X') \preceq_D c(x')$ uses $\mathcal{O}(d|X'|)$ comparisons.*

2.2 GRAPHS AND PATHS

If not specified otherwise, we consider directed graphs $G = (V, A)$ in which V is the set of nodes and $A \subseteq V \times V$ is the set of directed arcs. For an arc $(u, v) \in A$, we call u the *tail* of a and v the *head* of a. For a node $v \in V$, the set $\delta^-(v) \subseteq A$ denotes the set of arcs in A whose head node is v. For a node $u \in V$, we write $u \in \delta^-(v)$ if $(u, v) \in A$ exists, i.e. if $(u, v) \in \delta^-(v)$. Similarly, the set $\delta^+(v)$ denotes the set of arcs in A whose tail node is v and we write $w \in \delta^+(v)$ if $(v, w) \in A$ exists. As a convention, we set $n := |V|$ to be the number of nodes and $m := |A|$ the number of arcs in a given graph G.

Between two nodes s, $t \in V$, an s-t-*path* p is an ordered sequence of arcs. The tail of the first arc in p must be s and the head of the last arc of p must be t. For every arc but the last arc of p, the head of the arc must coincide with the tail of the next arc in p. For a node $v \in V$, we write $v \in p$ if v is the tail or the head node of an arc in p. An s-t-path p is *simple* if for every node $v \in p$, v has exactly one incoming and one outgoing arc in p. A path in which s and t are equal and no other nodes are repeated is called a *cycle*. Paths containing cycles are called *walks*.

Let p be a simple s-t-path and u, $v \in p$ two nodes. We refer to the u-v-subpath of p by $p^{u \to v}$. If p is an s-t-path and q is an t-r-path, the concatenation of both paths is an s-r-path and we refer to it by $p \circ q$. In the same way, we write $p \circ a$ for an arc $a \in \delta^+(t)$ if we consider the path obtained after adding the arc a to p. We denote the set of all s-t-paths in G by P_{st}.

Part II

LABEL SETTING ALGORITHMS FOR MULTIOBJECTIVE SHORTEST PATHS PROBLEMS

3 MULTIOBJECTIVE SHORTEST PATH PROBLEM

In this chapter we introduce the main problem studied in this thesis, the *Multiobjective Shortest Path* (MOSP) problem. It is a DMO problem and more specifically, a *Multiobjective Combinatorial Optimization* problem. We start the exposition with the formal definition of the problem in Section 3.1. Section 3.2 is about complexity of the problem. The results presented therein are not a contribution in this thesis; they can be found in the literature. In Section 3.4 we define *label-setting* algorithms for MOSP. This class of algorithms is known from the single-criterion Shortest Path problem. Defining it in a multiobjective context is important because the dominance order used to decide upon the optimality/efficiency of paths is a partial order. However, during the solution process of MOSP instances and particularly during label-setting algorithms, paths have to be sorted according to a total order until the algorithm can decide whether they are efficient or irrelevant. In Section 3.3 we discuss how to store paths with multidimensional cost vectors as *labels* in memory efficient MOSP algorithms. The interplay of the dominance order and the possibly used total orders is thus clarified in Section 3.4. One of the suitable total orders, the lexicographic order, stands out because it allows to use a technique called *dimensionality reduction* to speed up the $\preceq_D$-checks introduced in Definition 2.3. This technique was first used in a MOSP algorithm by Pulido et al. (2015). We describe it in Section 3.5. Finally, in Section 3.6 we discuss how we denote the different stages in which paths can be during label-setting MOSP algorithms. This introduces a disjoint partition of the set of paths in the considered digraph that is convenient for the description of the algorithms.

3.1 PROBLEM DEFINITION

We consider a directed graph and multiple arc cost functions. Using so called *sum objective functions* (cf. Ehrgott, 2005) these functions are extended to the paths in the considered graph.

Definition 3.1. Consider a directed graph $G = (V, A)$ and $d \in \mathbb{N}$ arc cost functions $c_i : A \to \mathbb{R}$, $i \in \{1, \dots, d\}$. For an arc $a \in A$, we write $c(a) := (c_1(a), \dots, c_d(a))$.

For nodes $s, v \in V$ let p be an s-v-path in G. Then the *costs of* p *w.r.t* c are

$$c(p) := \sum_{a \in p} c(a) \in \mathbb{R}^d. \tag{4}$$

We can now define the MOSP problem formally.

Definition 3.2 (Multiobjective Shortest Path Problem)**.** A d-*dimensional Multiobjective Shortest Path* (MOSP) instance consists of a digraph $G = (V, A)$, a designated source node $s \in V$, and $d \in \mathbb{N}$ arc cost functions $c_i : A \to \mathbb{R}$, $i \in \{1, \ldots, d\}$. Assume that the arc costs c are extended to the paths in G as in Definition 3.1.

ONE-TO-ONE Given an additional target node $t \in V$ in the input, the *One-to-One MOSP* problem (from s to t) is to find a minimal complete set P^*_{st} of s-t-paths in G. The tuple $\mathcal{I} := (D, s, t, d, c)$ is a One-to-One MOSP instance.

ONE-TO-ALL If no target node is given in the input, the *One-to-All MOSP* problem is to solve the One-to-One MOSP problem from s to v for every $v \in V \setminus \{s\}$. The tuple $\mathcal{I} := (D, s, d, c)$ is a One-to-All MOSP instance.

The MOSP problem variants with bidimensional instances are called One-to-One/One-to-All *Biobjective Shortest Path* (BOSP) problem.

In the remainder of the thesis if we do not specify if the considered MOSP instances are a One-to-One or a One-to-All MOSP instance, the statement holds for both types of instances. Moreover, we assume w.l.o.g. that every node $v \in V$ is reachable from s.

Remark 3.1. *Given a MOSP instance and a Shortest Path instance defined on the same graphs with the same source and possibly target node, the sets of feasible solutions coincide. In particular, the solution sets of One-to-All MOSP instances, can be characterized using arborescences in G (Raith & Ehrgott, 2009; Mote et al., 1991). For the scope of this thesis, this characterization is not required which is why in Definition 3.2 we define One-to-All MOSP instances as n One-to-One instances.*

In the single-criterion Shortest Path problem, arc cost functions that are not conservative can lead to the non-existence of an optimal finite path. A walk looping infinitely around a cycle with negative cost improves its cost after every loop. In a multiobjective setting, a single arc cost function that is not conservative suffices to have an infinite set of efficient paths. Let C be a cycle in G with $c_j(C) < 0$ for an index $j \in \{1, \ldots, d\}$. In all other dimensions let C have positive costs. Moreover, for nodes s, v in G, let p be an efficient s-v-path containing a node u that is also in C. For any $k \in \mathbb{N}$, define the sequence of paths

$$P = (p_1, \ldots, p_k), \quad p_j := p^{s \to u} \circ \underbrace{C \circ \ldots \circ C}_{j \text{ times}} \circ p^{u \to v}, \quad j \in \{1, \ldots, k\}.$$

Then, no two walks in P dominate each other and since p is efficient, all walks in P are also efficient. As a consequence, we consider MOSP instances without cycles with negative costs in the input graph.

Definition 3.3 (Conservative Arc Costs (cf. Definition 7.1 Korte & Vygen, 2005))**.** Let $G = (V, A)$ be a directed graph and $c : A \to \mathbb{R}$ an arc cost function. Then, c defines *conservative arc costs* if there is no cycle in G with negative total cost w.r.t. c.

If we consider MOSP instances with conservative arc costs only, minimal complete sets of efficient paths containing only simple paths exist. Note that conservative arc cost functions do not rule out the possibility of having negative arc cost components. However, it is well known that in the single-criterion case, when the arc cost function defines conservative arc costs, we can transform the arc costs in polynomial time w.r.t. the size of G to obtain nonnegative arc costs $\bar{c}$ w.r.t. which the ordering of the paths remains the same as w.r.t. the original costs c. This also holds in the multiobjective scenario (cf. Sedeño-Noda & Colebrook, 2019) and thus, we consider MOSP instances with nonnegative arc costs only.

In this setting if an efficient path contains a cycle C with $c(C) = 0$, a cycle-free representation of the path at hand exists and it has equivalent costs. In addition, we can derive the following structural property for efficient paths. It is a *Bellman condition*, often referred to as *subpath-optimality* in the single-criterion shortest paths literature.

Theorem 3.1 (Bellman condition for efficient paths. (E. Q. V. Martins, 1984, Lemma 4))**.** *Consider a d-dimensional MOSP instance with d conservative arc cost functions. For a node $v \in V$, let p be an efficient and simple s-v-path. Then, for every node $u \in p$, the s-u-subpath $p^{s \to u}$ of p is an efficient s-u-path w.r.t. c.*

Proof. Since p is a simple path, its s-u-subpath is well defined. We prove the statement by contradiction. If $p^{s \to u}$ is not an efficient s-u-path, there exists an s-u-path q that dominates it. As a direct consequence,

$$c(q \circ p^{u \to v}) = c(q) + c(p^{u \to v}) \prec c(p^{s \to u}) + c(p^{u \to v}) = c(p)$$

which contradicts the efficiency of p. □

As a direct consequence of the last theorem, we can formulate the following result.

Corollary 3.1. *Consider a d-dimensional MOSP instance with conservative arc cost functions and a node $v \in V$. For all nodes $u \in \delta^-(v)$ let P^*_{su} be a minimal complete set of efficient s-u-paths. Then, the set*

$$\left\{ p \circ (u,v) \;\middle|\; p \in P^*_{su},\ (u,v) \in \delta^-(v) \right\} \tag{5}$$

contains a minimal complete set of efficient s-v-paths.

From now on, we refer to d-*dimensional MOSP instances with conservative arc cost functions* simply as d-*dimensional MOSP instances*. Moreover, the following remark simplifies the proofs and the notation.

Remark 3.2 (Paths and Walks)**.** *The MOSP algorithms considered in this thesis generate paths arc by arc starting with an arc-less path at s. By doing so, we cannot guarantee that, given a simple s-v-path p for some $v \in V$, its expansion along an arc $(v,w) \in \delta^+(v)$ is a simple s-w-path $q = p \circ (v,w)$. In this scenario, let q' be the s-w-path obtained after removing the w-w-cycle from q.*

If we consider only MOSP instances with conservative arc costs, we have $c(q') \leqslant c(q)$. Hence, since paths and walks are built arc by arc, any MOSP algorithm considered in this thesis processes q' before q. Then, while building a minimal complete set of efficient s-w-paths, the walk q is dominated by or equivalent to q'. Hence, the conservative costs and the $\preceq_D$-checks ensure that walks are not contained in the returned minimal complete sets of efficient paths. Thus, in the upcoming chapters, we do not differentiate between paths and walks.

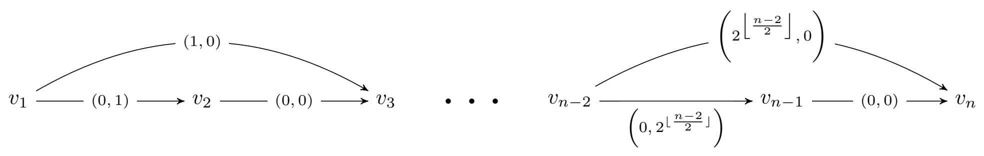

$(1,0)$
$\left(2^{\lfloor\frac{n-2}{2}\rfloor},0\right)$
v_1
$(0,1)$
v_2
$(0,0)$
v_3
v_{n-2}
v_{n-1}
$(0,0)$
v_n
$\left(0,2^{\lfloor\frac{n-2}{2}\rfloor}\right)$

Figure 2: BOSP instance by Hansen (1980). Every path is efficient.

3.2 HARDNESS, INTRACTABILITY, AND OUTPUT SENSITIVITY

Serafini (1987) proved that MOSP problems are $\mathcal{NP}$-hard. Recent discussions about the suitability of the notion of $\mathcal{NP}$-hardness for MOSP problems are conducted in (Bökler, 2018, 2017). Our contributions focus on the *intractability* of MOSP problems and the resulting output sensitive running time and space consumption bounds for MOSP algorithms.

MOSP instances in which every path is efficient first appeared in (Hansen, 1980). The author designed instances with two arc cost functions, i.e., BOSP instances. Since the number of paths in a graph is generally exponential w.r.t. the graph's input size, these instances prove the intractability of the problem. In Figure 2 we show one of these graphs.

Theorem 3.2 (Hansen (cf. 1980, Theorem 1))**.** *The MOSP problem is intractable.*

A consequence of intractability is that an exponentially sized output needs to be stored and returned by a MOSP algorithm. Even if the storage of efficient paths consumes $\mathcal{O}(1)$ memory per path, returning all of them is a linear time operation w.r.t. the number of efficient paths. As shown by Bökler (2018), the One-to-All MOSP problem is *output sensitive*. Indeed, the classical MOSP algorithm (E. Q. V. Martins, 1984) that we discuss in detail in Chapter 5 has an asymptotic running time bound that is polynomial w.r.t. the input and the output size of the given instance. In (Bökler, 2018) the author also proves that the One-to-One MOSP problem is not output sensitive unless $\mathcal{P} = \mathcal{NP}$. An intuitive reasoning is that the size of the output of a One-to-One MOSP algorithm, i.e., of a minimal complete set P^*_{st} of efficient s-t-paths can contain less paths than the efficient s-v-paths, $v \in V \setminus \{s, t\}$, that are needed to ensure that P^*_{st} is correct.

Theorem 3.3 (e.g., Breugem et al. (2017, Lemma 3.1.))**.** *The One-to-All MOSP problem is output sensitive.*

The main theoretical contribution in this part of the thesis is the so called *Multiobjective Dijkstra Algorithm*. Its output sensitive running time bound improves the lowest output sensitive running time bound for MOSP problems known so far.

3.3 LABELS

As mentioned earlier, we discuss MOSP algorithms that output minimal complete sets of efficient paths containing simple paths only. Thus, the longest possible output path w.r.t. the number of arcs contains $(n-1)$ arcs. *Labels* in a MOSP context are an abstract encoding of feasible paths that is memory efficient and that allows to reconstruct the encoded path in $\mathcal{O}(n)$ time. We assume that a label uses $\mathcal{O}(d)$ memory to encode the path it represents. In the single-criterion case labels use $\mathcal{O}(1)$ memory but in our scenario, we store the paths' cost vectors in the labels. Since a simple path p has $\mathcal{O}(n)$ s-v-subpaths, $v \in p$, storing p and all its subpaths using a label per subpath uses $\mathcal{O}(dn)$ memory. Due to the one-to-one correspondence between a path and the label that represents it, both notions can be used interchangeably.

Let $G = (V, A)$ be a digraph and p a simple s-v-path in G, $v \in V$. A label encoding p can be a tuple $\ell_p = (v, \mathrm{lastArc}_p, \mathrm{predLabel}_p, c(p))$ that contains the final node v of p, the incoming arc $\mathrm{lastArc}_p = (u, v)$ of v in p, a reference $\mathrm{predLabel}_p$ to the label encoding the $p^{s \to u}$ subpath of p, and the costs of p. Then, using Algorithm 1, p is reconstructed in $\mathcal{O}(n)$ time. Asymptotically it is safe to assume that the effort for rebuilding p vanishes in comparison with the time needed to compute ℓ_p in a label-setting MOSP algorithm.

Algorithm 1: Reconstruction of a path p given its label.

```
Input  : Digraph $G = (V, A)$, label
         $\ell_p = (v, \mathrm{lastArc}_p, \mathrm{predLabel}_p, c(p))$ encoding a simple
         $s$-$v$-path $p$ in $G$.
Output: $s$-$v$-path $p$
1 Path $p \leftarrow ()$;
2 Label $\ell \leftarrow \ell_p$;
3 while Node in $\ell$ is not $s$ do
4     Prepend arc in $\ell$ to $p$;
5     $\ell \leftarrow \mathrm{predLabel}$ stored in $\ell$;
6 return $p$.
```

If the costs of p are d-dimensional, ℓ_p as described above indeed uses $\mathcal{O}(d)$ space. Storing p itself, would require $\mathcal{O}(n)$ space since its arcs would need to be stored. Note that if needed, labels can store more information about the path they encode if the additional information requires $\mathcal{O}(d)$ memory. I.e., labels are implementation dependent data structures.

3.4 LABEL-SETTING ALGORITHMS FOR MOSP

In this section, we define *label-setting MOSP algorithms* as algorithms that exploit Corollary 3.1.

Definition 3.4 (Label-Setting MOSP Algorithm)**.** Consider a MOSP instance and let $\mathcal{A}$ be a MOSP algorithm and $v \in V$ be a node in G. A *label-setting MOSP algorithm* is a MOSP algorithm that only adds efficient paths to the set P^*_{sv} and only explores new s-v-paths by expanding efficient s-u-paths for $u \in \delta^-(v)$ along the incoming arcs (u, v) of v.

Thus, when a label-setting algorithm stores a path p in the output set P^*_{sv} it must be sure that a path that is stored later in this set does not dominate

p. To ensure this, we use the notion of *compatible orders*. Without giving the property a name, it has been already discussed in (Paixão & Santos, 2013).

Definition 3.5 (Compatible Orders)**.** Consider a d-dimensional MOSP instance and a total order $\prec_Q$ on the outcome space $c(P)$ induced by the set P of paths in G. $\prec_Q$ is called *compatible* with the dominance order (Definition 2.1) if for any $v \in V$ and any two s-v-paths p, q we have:

$$c(p) \prec_Q c(q) \Rightarrow c(q) \text{ does not dominate } c(p). \tag{6}$$

Example 3.1. Compatible Orders The following two total orders often used in the literature are compatible with the dominance order. Consider two vectors $x, y \in \mathbb{R}^d$, $d \in \mathbb{N}$.

LEXICOGRAPHIC ORDER We write $x \prec_{lex} y$ and say that x is lexicographically (lex.) smaller than y if for the first index $i \in \{1, \dots, i\}$ for which $x_i \neq y_i$, we have $x_i < y_i$.

SUM ORDER We write $x \prec_\Sigma y$ if $\sum_{i=1}^d x_i < \sum_{i=1}^d y_i$.

Proving that the orders in Example 3.1 are compatible with the dominance order is immediate. The sets of paths in a digraph used to define a MOSP instance inherits the $\prec_Q$ order from the instance's outcome space. The sum order is sometimes beneficial for a large number of objective vectors because the ordering of paths is achieved using comparisons of scalars only. However, this sorting key needs to be calculated for every path and does not outperform label-setting MOSP algorithms taking advantage of the benefits that come with the lexicographic ordering. Namely, it is particularly beneficial with regard to the $\preceq_D$-checks. These checks are, in theory and in practice, the main bottleneck in the performance of MOSP algorithms.

3.5 DIMENSIONALITY REDUCTION

A technique called *dimensionality reduction* can be used in conjunction with the lexicographic ordering of paths to reduce, in practice, the number of comparisons needed to answer $c(P) \preceq_D c(p)$ for a set of paths P and a path p. Dimensionality reduction in MOSP algorithms was first used in Pulido et al. (2015). The technique is not specific to paths which is why in this section we reproduce their exposition using d-dimensional vectors, $d \in \mathbb{N}$.

Consider a discrete set of vectors $X \subset \mathbb{R}^d$ that is sorted in $\prec_{lex}$-increasing order and assume that the vector $y \in \mathbb{R}^d$ is lex. greater than every element in X. By definition, we have $y_1 \geqslant x_1$ for every $x \in X$. Thus, to check if y is dominated by a vector in X we compare the 2nd to dth entries of the vectors.

Definition 3.6 (Dimensionality Reduced Front)**.** The *dimensionality reduced front* X_{dr} of X is the discrete set of $(d-1)$-dimensional vectors obtained by

1. Neglecting the first component of every vector in X and
2. deleting all dominated $(d-1)$-dimensional vectors obtained after the first step and keeping just one representative of equivalent cost vectors.

We always assume that the dimensionality reduced fronts are maintained in lex. increasing order. If $X_{dr} \subset R^{d-1}$ is such a front and we want to answer $X_{dr} \preceq_D x$ for a vector $x \in \mathbb{R}^{d-1}$, we only need to compare x with the vectors in X_{dr} that are not lex. greater than x. This yields big running time improvements in practice.

Example 3.2. Consider the set $X = \{(1,2,3),\ (3,2,1),\ (2,1,3)\}$. To build X_{dr}, we consider the set $\{(2,3),\ (2,1),\ (1,3)\}$ obtained after neglecting the first component of the original vectors. Since $(2,3)$ is dominated by the other vectors in this set, it is discarded. Finally after re-sorting the remaining vectors, we obtain $X_{dr} = \{(1,3),\ (2,1)\}$.

The following theorem explains why dimensionality reduced fronts are relevant in practice. Its proof is straightforward and can be found in the original publication.

Lemma 3.1 (Pulido et al. (2015, Lemma 2)). *Let $X \subset \mathbb{R}^d$ be a discrete set and $y \in \mathbb{R}^d$ a vector that is lex. greater than every vector in X. Then, we have*

$$X \preceq_D y \Leftrightarrow X_{dr} \preceq_D (y_2, \ldots, y_d). \tag{7}$$

Running Time

Because of the removal of elements in Item 2 of the enumeration in Definition 3.6, there holds $|X_{dr}| \leqslant |X|$ but, the inequality can be tight. Thus, even though the vectors in X_{dr} are $(d-1)$-dimensional, the technique cannot be used to reduce the asymptotic running time of the $\preceq_D$-check derived in Proposition 2.1. The number of comparisons to solve $X_{dr} \preceq_D (y_2, \ldots, y_d)$ is in $\mathcal{O}\left((d-1)|X|\right) = \mathcal{O}\left(d|X|\right)$ for any $d \geqslant 3$.

Assume X_{dr} is sorted in lex. non-decreasing order and we need to possibly update X_{dr} to contain the vector $x \in \mathbb{R}^{d-1}$. In order for $X_{dr} \cup \{x\}$ to be a dimensionality reduced front and remain sorted after x's possible insertion, we need to make sure that x is not dominated by or equivalent to a vector in X_{dr}, find the correct insert position for x and delete vectors in X_{dr} that are dominated by x. This operation is called a *merge* operation. It is used extensively in classical label-setting MOSP algorithms in a different context than dimensionality reduction.

Definition 3.7 (Merge Operation). In any dimension $d \geqslant 2$, consider a discrete set $X \subset \mathbb{R}^d$, a vector $x \in \mathbb{R}^d$, and a total order $\prec_Q$ on $\mathbb{R}^d$ that is compatible with the dominance order (cf. Definition 3.5). Assume that the elements in X are sorted in ascending order w.r.t. $\prec_Q$. A *merge operation* (Algorithm 2) has three purposes:

1. Answer $X \preceq_D x$, i.e., whether there is an element in X that dominates x or is equivalent to x. If not,
2. insert x into X s.t. X remains sorted in ascending order w.r.t. $\prec_Q$.
3. Finally, remove elements in X that are dominated by x.

Algorithm 2 contains the pseudocode of a merge operation. We refer to it using the merge link from now on. Note that X is a linked list in the pseudocode and thus, accessing its i^{th} element using $X[i]$ is not a constant time operation. From the pseudocode we can immediately see the following complexity result.

Algorithm 2: merge

```
Input  : Discrete set $X \subset \mathbb{R}^d$ implemented as a doubly linked list, sorted
         in ascending order w.r.t. $\prec_Q$, and indexed from $0$ to $|X|-1$. Vector
         $x \in \mathbb{R}^d$.
Output: Updated set $X$ according to Definition 3.7.
1  Index $i \leftarrow 0$;
2  Vector $y \leftarrow X[i]$;                  // $[\cdot]$ accesses the $i$th vector in the list.
3  while $i \neq |X|-1$ and $y \preceq_Q x$ do
4      if $y \preceq_D x$ then return $X$ ;
5      $i \leftarrow i+1$;
6      $y \leftarrow X[i]$;
7  $X \leftarrow X$ with $x$ inserted in position $i$;
8  while $i \neq |X-1|$ do
9      if not $x \preceq_D y$ then
10         $i \leftarrow i+1$;
11         $y \leftarrow X[i]$;
12     else
13         $X \leftarrow$ remove $y$ from $X$;  // Don't update $i$. $X[i]$ is now the successor
             of $v$ in the original $X$.
14 return $X$;
```

Proposition 3.1. *Algorithm 2 runs in $\mathcal{O}(d|X|)$.*

Recall that in the context of the dimensionality reduction technique, we require the total order $\prec_Q$ in Definition 3.7 to be the $\prec_{lex}$ order. We conclude that the addition of $x \in \mathbb{R}^{d-1}$ to a dimensionality reduced front $X_{dr} \subset \mathbb{R}^{d-1}$ s.t. $X_{dr} \cup \{x\}$ is a sorted dimensionality reduced front is done in $\mathcal{O}(d|X_{dr}|)$ comparisons using merge operations.

Dimensionality Reduction in Biobjective Shortest Path Problems

The greatest impact of performing dominance tests using the right hand side of (7) is achieved if the original vectors in X are bidimensional.

Proposition 3.2. *Let X be a discrete set in $\mathbb{R}^2$. Then, X_{dr} is a single number.*

Corollary 3.2. *Let X be a discrete set in $\mathbb{R}^2$ and $y \in \mathbb{R}^2$ a vector that is not lex. smaller than any vector in X. Then, $X \preceq_D y$ is checked in $\Theta(1)$.*

Our implementations of label-setting MOSP algorithms always use the dimensionality reduction to obtain best possible running times in practice. The impact of the technique is assessed in the benchmarks in the original publication (Pulido et al., 2015). However, since the use of the dimensionality reduction technique does not improve the asymptotic running time bound of the $\preceq_D$-checks in general dimensions, we do not include it in our algorithmic discussions in the next two chapters. As an additional benefit, we can describe the label-setting algorithms in this part of the thesis using an unspecified $\prec_Q$ order that is compatible with the dominance order.

$$\begin{tikzcd}
 & & \text{permanent} \\
\text{unexplored} \arrow[r] & \text{explored} \arrow[ur] \arrow[dr] & \\
 & & \text{discarded}
\end{tikzcd}$$

Figure 3: For any $v \in V$ every s-v-path is in one of these sets during a label-setting algorithm (cf. Definition 3.8). If a path reaches the set of permanent paths or the set of discarded paths, it stays there until the end of the algorithm. At the end of the algorithm, the set of explored paths is empty.

3.6 PARTITION OF THE SET OF PATHS

We assume that any label-setting MOSP algorithm maintains a list P^*_{sv} of s-v-paths for every $v \in V$. At the beginning of the algorithm, these lists are empty. Exploiting the label-setting property of the algorithm, we further assume that only efficient s-v-paths are stored in P^*_{sv}. In a One-to-All MOSP algorithm these sets are the algorithms' output and are thus proven to then contain a minimal complete set of efficient s-v-paths. In a One-to-One scenario this property does not necessarily hold and only P^*_{st} is a minimal complete set of efficient s-t-paths.

Label setting MOSP algorithms implicitly partition the set of paths in G. See also Figure 3.

Definition 3.8 (Paths Partition)**.** Consider a MOSP instance and a label-setting MOSP algorithm $\mathcal{A}$. Then, $\mathcal{A}$ implicitly partitions the set of paths in G in four disjoint sets.

UNEXPLORED PATHS This set contains every path in G that has not been processed by the algorithm.

PERMANENT PATHS A path $p \in \bigcup_{v \in V} P^*_{sv}$ is a permanent path. I.e., permanent paths are the efficient paths already computed and stored by the algorithm.

DISCARDED PATHS An s-v-path p for some $v \in V$ is a discarded path if it is not an unexplored path and if $c(P^*_{sv}) \preceq_D c(p)$.

EXPLORED PATHS This set contains every path that is neither of the last three sets of paths. In other words, a path is an explored path if $\mathcal{A}$ has processed it already but could not yet determine whether it is a permanent path or a discarded path. Note that the list P^*_{sv} is modified with every efficient s-v-path that is found. Thus, the check $c(P^*_{sv}) \preceq_D c(p)$ to determine whether an explored s-v-path p is discarded or made permanent needs to be performed in the *right moment*. At the end of a label-setting algorithm, the set of explored paths is empty.

The partition of paths is implicit because neither unexplored paths nor discarded paths are stored. Label setting algorithms clearly maintain the sets P^*_{sv} with the labels encoding the efficient s-v-paths found so far. On one hand and particularly in a One-to-All scenario these sets are the algorithms' output. On the other hand, the storage of these sets is needed to take advantage of Corollary 3.1: once an s-v-path p is stored in P^*_{sv}, it can be expanded along the outgoing arcs of v to generate new candidate s-w-paths for $w \in \delta^+(v)$. These new paths are explored paths until they are either made permanent or discarded. The handling of explored paths is a key issue in the development of efficient MOSP algorithms. MOSP algorithms existing before the publications that are reproduced in this thesis, stored explored paths explicitly. Our new Multiobjective Dijkstra Algorithm (see Chapter 4) accentuates the benefits of the Bellman condition for MOSP (Theorem 3.1) to achieve better asymptotic bounds.

4 MULTIOBJECTIVE DIJKSTRA ALGORITHM

In this chapter we introduce the *Multiobjective Dijkstra Algorithm* (MDA). The algorithm is the main contribution in this thesis. As discussed initially, the algorithm is a One-to-All MOSP algorithm. Out of the box it then also solves a One-to-One MOSP instance defined on the same graph. Already in this chapter we discuss on a high level some improvements to enhance the MDA's performance when applied to One-to-One MOSP instances. Chapter 9 is devoted completely to the design of a tuned One-to-One version of the MDA.

The remaining contributions in this thesis build upon the base version of the MDA. The pseudocode is given in Algorithm 3, which we link to using the MDA hyperlink throughout the thesis. The MDA is, to the best of our knowledge, the first label-setting MOSP algorithm that handles explored paths implicitly. In particular, at any point in time during its execution and for any node $v \in V$, it stores at most one explored path. When the stored s-v-path for a node $v \in V$ is either made permanent or discarded, a new explored path for v is generated out of the permanent s-u-paths for $u \in \delta^-(v)$, taking advantage of Corollary 3.1.

For the description of the algorithm, we assume that a d-dimensional One-to-All MOSP instance $\mathcal{I} = (D, s, d, c)$ is given. The data structures used in the MDA are a priority queue Q of paths that are sorted in nondecreasing order w.r.t. a total order $\prec_Q$ that is compatible with the dominance order and the lists P^*_{sv} of s-v-paths for every $v \in V$.

SHORT DESCRIPTION In every iteration, the MDA extracts a $\prec_Q$-minimal path p from Q (Line 7). Let p be an s-v-path for a node $v \in V$. It is guaranteed that p is not dominated by and not cost-equivalent to any path already stored in P^*_{sv}. Thus, p is made permanent, i.e., added to P^*_{sv} (Line 8). Then, new explored s-w-paths for $w \in \delta^+(v)$ are generated by concatenating p with the outgoing arcs of v (Line 9). We discuss later how the new explored paths are handled. Possibly, some of them are added to Q. Since the priority queue Q contains at most one s-v-path at any point in time, it has no s-v-path after p is extracted. Hence, in the current iteration, other previously explored s-v-paths are reconsidered. If it exists, a minimal one w.r.t. $\prec_Q$ that is neither dominated by nor cost-equivalent to a path already stored in P^*_{sv} is inserted into Q (Line 10 and Line 11). The MDA terminates when the priority queue Q is empty at the beginning of an iteration.

4.1 MDA, PROPAGATE, AND NEXTQUEUEPATH

We proceed with a detailed description of the MDA and its subroutines. The main novelty is the handling of explored paths that are not stored in the priority queue Q.

The datastructures of the algorithm are initialized in Line 1-Line 5. Initially, they are all empty. The trivial path from s to itself with costs $0 \in \mathbb{R}^d$ is added to the priority queue Q and is the only explored path before the first iteration of the algorithm's main loop. We call this path p_{init} and initialize it in Line 4.

Algorithm 3: Multiobjective Dijkstra Algorithm (MDA)

```
Input  : d-dimensional One-to-All MOSP instance $(G, s, d, c)$.
Output : Minimal complete set $P^*_{sv}$ of efficient $s$-$v$-paths for every
         $v \in V$.
1  Priority queue of paths $Q \leftarrow \emptyset$;            // Sorted according to $c$.
2  $\forall (u,v) \in A$ – explored $s$-$v$-paths that use $(u,v)$ as last arc: $\mathrm{lpl}_{uv} \leftarrow \emptyset$;
3  $\forall v \in V$ – permanent $s$-$v$-paths: $P^*_{sv} \leftarrow \emptyset$ ;
4  $p_{\text{init}} \leftarrow ()$ ;
5  $Q \leftarrow Q.\text{insert}(p_{\text{init}})$;
6  while $Q \neq \emptyset$ do
7      $s$-$v$-path $p \leftarrow Q.\text{extractMin}()$;
8      $P^*_{sv} \leftarrow$ Append $p$ to $P^*_{sv}$;
9      $Q \leftarrow$ propagate$(p, Q, P^*_{sw}$ for every $w \in \delta^+(v))$
10     $s$-$v$-path $p^{\text{new}} \leftarrow$ solve (8) and if no solution is found, set $p^{\text{new}}$ to
        NULL ;
11     if $p^{\text{new}} \neq$ NULL then $Q.\text{insert}(p^{\text{new}})$;
12 return $\cup_{v \in V} P^*_{sv}$;
```

As noted in the short description of the MDA, for every node $v \in V$, the priority queue Q contains at most one s-v-path at any point in time. We thus refer to *v's queue path* and mean the s-v-path in Q if one exists. In every iteration the MDA performs three tasks.

EXTRACTION AND STORAGE A minimal (w.r.t. $\prec_Q$) path p is extracted from Q in Line 7. Assume that p is an s-v-path, $v \in V$. It is guaranteed that p is an efficient path and thus, it is added to P^*_{sv} in Line 8.

EXPLORING NEW PATHS Algorithm 4 is called in Line 9 of the MDA. The call is triggered in every iteration. The subroutine is called *propagate* and linked using the propagate link. In it, the extracted s-v-path p is expanded along the arcs $(v,w) \in \delta^+(v)$. This produces new explored s-w-paths. Let $q = p \circ (v,w)$ be such a path (Line 2 of propagate). If q is neither dominated by nor equivalent to a path in P^*_{sw} (Line 3 of propagate), two cases are distinguished:

- If there is no s-w-path in Q, q is inserted into Q (Line 8 of propagate of propagate).

- If there is an s-w-path q' in Q and q' is greater than q w.r.t. $\prec_Q$, q replaces q' in Q (Line 5 of propagate).

In any other case, q is ignored. The MDA considers it again later to either discard it or make it w's queue path. Hence, q is now an explored path in the MDA but it might not stored. It can be restored efficiently later.

Algorithm 4: propagate.

```
Input  : $s$-$v$-path $p$, priority queue $Q$,
         $s$-$w$-paths $P^*_{sw}$ for every $w \in \delta^+(v)$.
Output: Updated priority queue $Q$.
1 for $(v, w) \in \delta^+(v)$ do
2     $s$-$w$-path $q \leftarrow p \circ (v, w)$;
3     if $c(P^*_{sw}) \preceq_D c(q)$ then continue;
4     if $Q$ contains an $s$-$w$-path $q'$ then
5         if $c(q) \prec_Q c(q')$ then Replace $q'$ with $q$ in $Q$ ;
6     else
7         Insert $q$ into $Q$;
8 return $Q$;
```

FINDING NEXT QUEUE PATHS After p's extraction from Q at the beginning of the iteration (Line 7 of the MDA), there is no s-v-path in Q. Thus, the MDA needs to rebuild explored s-v-paths to possibly find a new queue path for v. The set of candidate s-v-paths consists of s-v-paths that have been already processed in calls to propagate in earlier iterations. These paths were built in Line 2 of propagate as the extension of an s-u-path for some $u \in \delta^-(v)$. In the corresponding iteration of the MDA, this s-u-path was extracted from Q and stored in P^*_{su}. Thus, in the current iteration of the MDA the next queue path for v is searched among the permanent s-u-paths for $u \in \delta^-(v)$. The candidate paths must not be dominated by or cost-equivalent to paths already stored in P^*_{sv}. All in all, a new queue path for v is an element in

$$\arg\min_{\prec_Q} \left\{ c(q \circ (u, v)) \mid q \in P^*_{su},\ (u, v) \in \delta^-(v),\ \text{not } c(P^*_{sv}) \preceq_D c(p) \right\}. \tag{8}$$

Thus, w.r.t. to the total order of the paths induced by $\prec_Q$, the new path is the most promising path to become a new permanent s-v-path. The compatibility of $\prec_Q$ with the dominance relation ensures that this path is not dominated by $\prec_Q$-greater ones. Finally if (8) is nonempty, the new s-v-path becomes the new queue path for v and is inserted into Q in Line 11 of the MDA. In Section 4.3 we discuss how to solve (8) efficiently in every iteration of the MDA.

If the queue Q of the MDA is empty at the beginning of an iteration, no new permanent paths can be extracted and the algorithm ends. Its output is the union of all sets P^*_{sv} for $v \in V$ and we claim that these sets are a minimal complete set of s-v-paths for every $v \in V$.

4.2 CORRECTNESS

First note that in the MDA unexplored paths become explored paths (cf. Figure 3) in Line 2 of propagate only. To solve (8) in Line 10 of the MDA the algorithm also reconstructs explored s-v-paths $p = q \circ (u, v)$ but such paths p have already become explored paths in the iteration of the MDA in which q was extracted from Q and added to P^*_{su}. Even though not storing the set of explored paths explicitly is one of the main innovations of the MDA, in this section we need to characterize, for every $v \in V$, the set E_{sv} of explored s-v-paths. E_{sv} is, precisely, the set considered to solve (8). The next lemma explains the interplay between the queue path for v and the paths in the list E_{sv} at the end of an iteration of the MDA.

Lemma 4.1. *Fix a node $v \in V$. At the end of an iteration of the MDA there are two options. If there is an s-v-path p in Q, then p is minimal w.r.t. $\prec_Q$ among the paths in E_{sv}. Moreover, E_{sv} is empty iff there is no queue path from v.*

Proof. Assume p is an s-v-path in Q at the end of an iteration of the MDA. p is inserted into Q in Line 11 of the MDA, in Line 8 of propagate, or in Line 5 of propagate. In any case, the insertion is preceded by a $\preceq_D$-check that guarantees that p is neither dominated by nor cost-equivalent to a path in P^*_{sv}.

Let p' be the queue path for v before p was inserted into Q. If p is inserted into Q in Line 11 of the MDA, then p' was extracted from Q at the beginning of this iteration and thus, p enters Q as a solution to (8). This proves the statement for this scenario.

Otherwise if p replaces p' in Q after a decrease key operation triggered in Line 5 of propagate, we have $c(p) \prec_Q c(p')$. Applying the arguments from this proof recursively until the first time an s-v-path is inserted into Q during the MDA and using the transitivity of the total order $\prec_Q$ we can then conclude that p is $\prec_Q$ minimal among the paths in E_{sv}.

Proving that E_{sv} is empty at the end of an iteration of the MDA iff there is no queue path for v in Q, uses the same arguments. □

As a consequence, we formulate the following result.

Lemma 4.2. *Let p be the path extracted from Q at the beginning of the i^{th} iteration of the MDA for some $i \in \mathbb{N}$ and let v be its end node. Moreover if a solution to (8) exists in the i^{th} iteration of the MDA , call it p^{new}. Then, $c(p) \prec_Q c(p^{new})$.*

Proof. Before it is made permanent, p is an explored s-v-path and thus in E_{sv}. By Lemma 4.1, we know that at the end of the $(i-1)^{th}$ iteration, p is a $\prec_Q$-minimal path in E_{sv}. After p is added to P^*_{sv} it becomes a permanent path. Thus, (8) is solved over the set E_{sv} that no longer contains p. If the result p^{new} would be $\prec_Q$-smaller than p, p would not have been $\prec_Q$-minimal. Since a solution to (8) must also not be cost-equivalent to a path in P^*_{sv}, we have $c(p) \prec_Q c(p^{new})$. □

We stick to the consideration of s-v-paths for a fixed node $v \in V$ for now. As so far, let p be such a path. Assume that p is contained in P^*_{sv} at the end of the MDA. Then, p is extracted from Q in Line 7 of the MDA at the beginning of an iteration. A particular behavior of the MDA is that once p is inserted into Q, it can be replaced before its final extraction. This replacement happens in Line 5 of propagate. If it happens, p is reinserted into Q in a later iteration as a solution to (8). This explains why in the proof of the next lemma and also in other results in this chapter, we refer to *the last time* p *is inserted into* Q. The next lemma is a direct consequence of the two previous ones.

Lemma 4.3. *Let* p *be an* s-v-*path that is made permanent during the* MDA. p *is neither dominated by nor equivalent to a path previously appended to* P^*_{sv}.

Proof. Before p is inserted into Q for the last time after its extraction from Q in Line 7 of the MDA, it is inserted into Q in propagate or after nextQueuePath. Before every insertion, $c(P^*_{sv}) \preceq_D c(p)$ is checked. Since p is the queue path for v in Q until its extraction, no other s-v-path is made permanent after $c(P^*_{sv}) \preceq_D c(p)$ is checked. Thus, after its extraction, p is still neither dominated by nor equivalent to a path in P^*_{sv}. □

Now, to prove that the MDA is a label-setting algorithm and to derive its correctness, we need to prove that paths added to P^*_{sv} after p do not dominate p and are not cost-equivalent to p. Lemma 4.2, paired with the compatibility of the $\prec_Q$ order with the dominance order, gives a strong hint already. For the formal proof however we first need to prove the following lemma that does not consider s-v-paths only.

Lemma 4.4. *Let* p *and* q *be two paths extracted from* Q *in Line* 7 *of the* MDA. *If* p *is extracted in an earlier iteration than* q, *then* $p \preceq_Q q$.

Proof. We prove the statement by induction over the iterations of the MDA. We take advantage of the nonnegativity of the arc cost functions and of the transitivity of $\prec_Q$.

At the beginning of the first iteration Q contains only the trivial path p from s to s with 0 cost. New paths are created and inserted into Q during propagate and for each such path q we have $c(p) \preceq_Q c(q)$. Thus, the path extracted from Q at the beginning of the second iteration of the MDA is not $\prec_Q$-smaller than p.

Let the statement hold until the k^{th} iteration for some $k \in \mathbb{N}$. Let p be the path extracted from Q at the beginning of the iteration and $v \in V$ be its end node. Any path in Q is not smaller than p w.r.t. $\prec_Q$ since Q is ordered and p is a minimal element. New explored paths q generated in propagate and possibly added to Q are concatenations of p along outgoing arcs of v and hence $c(p) \preceq c(q)$.

The only further insertion of a path to Q during the k^{th} iteration is done in Line 11 of the MDA after solving (8). By Lemma 4.2 the path inserted into Q after solving (8) in the k^{th} iteration is $\prec_Q$-greater than the path extracted from Q in this iteration. All in all if p is a $\prec_Q$-minimal path in Q when it is extracted and in the same iteration only paths that are not $\prec_Q$-smaller than p are inserted into Q, the extracted path in the $(k+1)^{th}$ iteration of the MDA is not $\prec_Q$-smaller than p. □

Since the lists of permanent paths in the MDA are only modified in Line 8 of the algorithm by appending paths at the lists' end, we can conclude from the last lemmas that the paths in the lists are ordered increasingly w.r.t. $\prec_Q$. Finally, we use the compatibility of $\prec_Q$ with the dominance order to prove the label-setting property of the MDA.

Lemma 4.5. *Let $v \in V$ be a node and consider the list P^*_{sv} of permanent s-v-paths in the MDA. The paths in P^*_{sv} are sorted in $\prec_Q$-increasing order and a new permanent path does not dominate any path already in P^*_{sv} and is also not cost-equivalent to a path in P^*_{sv}.*

Proof. Paths are stored in P^*_{sv} after their extraction from Q in Line 7 of the MDA. The storage happens by appending new paths to the list. Thus, as a direct consequence from Lemma 4.4 we know that the paths in P^*_{sv} are stored in $\prec_Q$ non-decreasing order. If we consider Lemma 4.2 additionally, we know that no two paths in P^*_{sv} are cost-equivalent and thus, we can conclude that the paths in the list are sorted in strict increasing order w.r.t. $\prec_Q$. At any stage of the MDA, let p and q be two paths in P^*_{sv} and assume that p was appended to the list before q. Then, the compatibility of $\prec_Q$ and the dominance order (Definition 3.5) ensures that q does not dominate p. Moreover, the $\preceq_D$-checks performed before paths are inserted into Q also exclude the possibility of q being cost-equivalent to p, which proves the statement. □

Pairing Lemma 4.3 and Lemma 4.5 we know that paths in P^*_{sv} for any $v \in V$ do not dominate each other and are not cost-equivalent. Finally, the correctness of the MDA follows proving that at the end of the algorithm for every $v \in V$ and for every nondominated cost vector in $c(P_{sv})$ there is an efficient path in P^*_{sv}.

Theorem 4.1. *Consider a node $w \in V$ and let P_{sw} be the set of all simple s-w-paths in G. For every nondominated point in $c(P_{sw})$ there is a path in the list P^*_{sw} of s-w-paths output by the MDA.*

Proof. We prove the statement by contradiction. Assume that for $w \in V$ $\gamma \in c(P_{sw})$ is a nondominated point and that there is no s-w-path p in P^*_{sw} with $c(p) = \gamma$. Consider the subset $P_\gamma \subseteq P_{sw}$ containing all simple s-w-paths with cost γ and fix a path $p \in P_\gamma$ for the remainder of the proof. Since p is an efficient path, the Bellman condition in Theorem 3.1 guarantees that for every node $u \in p$, the s-u-subpath $p^{s\to u}$ of p is an efficient s-u-path. Let u be the last node along p for which the s-u-subpath $p^{s\to u}$ of p is contained in P^*_{su} at the end of the MDA. The extreme case is $u = s$. Let v be the successor node of u along p, i.e., $(u, v) \in p$. During the expansion of $p^{s\to u}$ along (u, v) in Line 2 of propagate or while solving (8) the subpath $p^{s\to v}$ is ignored. Given that $p^{s\to v}$ is an efficient s-v-path, the reason must be that during a $\preceq_D$-check, a cost-equivalent s-v-path q is found in P^*_{sv}. Note that $r = q \circ p^{v\to w}$ is an efficient s-w-path and $r \in P_\gamma$. Thus, by assumption, it is not in P^*_{sw}.

We follow, arc by arc, the expansions of q along $p^{v\to w}$. Since r is efficient, they are all efficient paths by Theorem 3.1. Let $r^{s\to v'} = q \circ p^{v\to v'}$ be the first subpath of r that is not made permanent, i.e., that is not stored in $P^*_{sv'}$. Again, it is ignored because a cost-equivalent s-v′-path p′ exists in $P^*_{sv'}$.

We have progressed at least one arc along $p^{u\to w}$ and have proven the existence of a path $p' \in P^*_{sv'}$ that is cost-equivalent to $p^{s\to v'}$. After repeating the same argument at most $(n-1)$ time (because we consider simple paths) we obtain an explored s-w-path $p'' \in P_\gamma$ considered in Line 2 of propagate and possibly while solving (8). Since p'' is efficient and $P^*_{sw} \cap P_\gamma$ is empty, the check $c(P^*_{sw}) \preceq_D c(p'')$ does not discard p''. □

The final result in this section is now easy to prove.

Theorem 4.2. *At the end of the MDA, the list P^*_{sv} of s-v-paths output by the algorithm is a minimal complete set of efficient s-v-paths.*

Proof. By Theorem 4.1 for every $v \in V$ the set P^*_{sv} contains at least one efficient path for every non dominated point in $c(P_{sv})$. Lemma 4.5 guarantees that the paths in P^*_{sv} are sorted in monotonically increasing order w.r.t. $\prec_Q$ which implies that no two paths are cost-equivalent. Thus, for every non dominated point in $c(P_{sv})$ there is one path in P^*_{sv}. Finally, Lemma 4.3 and Lemma 4.5 also state that no two paths in P^*_{sv} dominate each other, ensuring that no dominated paths are output by the MDA. □

4.3 CALCULATING NEXT QUEUE PATHS

To prove the correctness of the MDA it was not necessary to discuss how to solve (8) in Line 10 of the MDA efficiently. We close this gap in this section. We call this step, performed in every iteration of the MDA, the calculation of a node's *next queue path*. Recall that the set E_{sv} is not stored implicitly.

How to compute a node's next queue path has a big impact on the MDA's asymptotic and computational running time. If we recall Figure 3, we also realize that we have not yet discussed when an explored path is finally discarded by the MDA. This question is also answered in this section.

Let $v \in V$ be a node. Solving (8) in a naive way in every iteration in which a new queue path for v is needed requires to rebuild explored s-v-paths repeatedly. But we know that the list P^*_{sv} is built incrementally appending new permanent paths at its end and never deleting paths from it. Thus if for an explored s-v-path p we have $c(P^*_{sv}) \preceq_D c(p)$, this holds until the end of the algorithm and thus p does not need to be further considered, i.e., it can be discarded.

Let N_v be the cardinality of a minimal complete set of efficient s-v-paths. At the end of the MDA we have $|P^*_{sv}| = N_v$ and the MDA solves (8) for node v exactly N_v times. Let i and j, $i < j$, be the indices of two iterations of the MDA in which an s-v-path is extracted from Q and assume that between both iterations no s-v-path is extracted from Q. We refer by E^i_{sv} and E^j_{sv} to the sets of s-v-paths built to solve (8) in Line 10 of the MDA in the i^{th} and in the j^{th} iteration, respectively. Every path $p \in E^i_{sv}$ s.t. $c(P^*_{sv}) \preceq_D c(p)$ can be ignored when building E^j_{sv}.

Moreover, for every $u \in \delta^-(v)$ the paths in P^*_{su} are sorted in $\prec_Q$-increasing order by Lemma 4.5. Since they are all expanded along the arc (u,v), their concatenations with the arc (u,v) are sorted equally. Let $p \in P^*_{su}$ be a permanent s-u-path and $q = p \circ (u,v)$ its expansion along (u,v). If q is neither dominated by nor equivalent to a path in P^*_{sv}, i.e. if $c(P^*_{sv}) \preceq_D c(p \circ (u,v))$

does not hold, q is a candidate to be a solution to (8). Since (8) is a minimization problem w.r.t. $\prec_Q$, every path that comes after p in P^*_{su} yields a worse candidate s-v-path. Hence, while solving (8) the first path in P^*_{su} that yields an expansion along (u, v) that is neither dominated by nor equivalent to a path in P^*_{sv} yields the only needed solution candidate for this incoming arc of v. The expansions of paths that come after p in P^*_{su} are considered in later iterations of the MDA.

The considerations in the last two paragraphs can be used to design a solution algorithm to (8) that considers a reduced range of candidates paths as opposed to an algorithm that just considers every path in P^*_{su} for all $u \in \delta^-(v)$. The pseudocode of this algorithm is given in Algorithm 5. We call it nextQueuePath.

Algorithm 5: Algorithm nextQueuePath to solve (8)

```
Input  : Node $v$, Indices $\text{lastProcessedPath} \in \mathbb{N}^{|A|}$, Permanent
         paths $P^*_{su}$ for $u \in \delta^-(v)$, Permanent paths $P^*_{sv}$.
Output: Solution to (8) if one exists and updated indices
         $\text{lastProcessedPath}$.

1  $s$-$v$-path $p \leftarrow$ NULL with $c(p) \leftarrow (\infty, \dots, \infty) \in \mathbb{R}^d_{\geqslant 0}$;
2  for $u \in \delta^-(v)$ do
3      if $P^*_{su}$ is empty then continue;
4      for $k \in [\text{lastProcessedPath}[(u,v)], |P^*_{su}|]$ do
5          $s$-$v$-path $q \leftarrow$ expansion of the $k^{\text{th}}$ path in $P^*_{su}$ along $(u, v)$;
6          $\text{lastProcessedPath}[(u,v)] \leftarrow k$ ;
7          if $c(P^*_{sv}) \preceq_D c(q)$ then
8              continue;
9          else
10             if $c(q) \prec_{\text{lex}} c(p)$ then $p \leftarrow q$ ;
11             break;
12 return $p$ and $\text{lastProcessedPath}$;
```

The algorithm requires that in the initialization of the MDA we allocate a vector of *last processed paths* indexed by the arcs in the input graph. I.e., for every arc $(u, v) \in A$, the vector lastProcessedPath has an entry. The entries of the vector are integers and they are initialized to 1. They are increased by one in Line 6 of nextQueuePath. This is the only modification of the indices. The following invariants explain the meaning of the entry $\text{lastProcessedPath}[(u,v)]$ during the MDA.

Invariant 4.1. *For any arc $(u, v) \in A$, there holds:*

RANGE $\text{lastProcessedPath}[(u,v)] \in \{1, \dots |P^*_{su}|\}$.

INTERPRETATION *Assume* $\text{lastProcessedPath}[(u,v)] = j > 1$ *at the beginning of an iteration of the MDA. For any $i \in \{1, \dots, j-1\}$ let p be the i^{th} path in P^*_{su} and $q = p \circ (u, v)$. Then, the dominance or equivalence check $c(P^*_{sv}) \preceq_D c(q)$ in Line 7 of nextQueuePath has been answered positively in a previous call of the MDA to nextQueuePath with v as its input. I.e., for $i < j$, the expansion of the i^{th} path in P^*_{su} along (u, v) is a discarded s-v-path.*

Proof. The range invariant holds since the lastProcessedPath indices are updated to the value of k in Line 6 of nextQueuePath and k is not greater than the cardinality of the set P^*_{su}.

We prove the second invariant. k is initialized to $\text{lastProcessedPath}[(u,v)]$ at the beginning of the inner loop of nextQueuePath. This loop has two possible outcomes: either a new iteration is triggered by the **continue** statement in Line 8 or the loop is stopped by the **break** statement in Line 11. The **continue** statement is reached if $c(P^*_{sv}) \preceq_D c(q)$, where $q = p \circ (u,v)$ and p is the k^{th} permanent path in P^*_{su}. Then, the next iteration of the inner loop starts and k's value is increased by one. In every iteration the value of $\text{lastProcessedPath}[u,v]$ is set to the new value of k (Line 6). Hence, let i be the value of $\text{lastProcessedPath}[(u,v)]$ at the beginning of a call to nextQueuePath and j be the value of $\text{lastProcessedPath}[(u,v)]$ at the end of this call to nextQueuePath. The expansions of all paths in the i^{th} to $(j-1)^{\text{th}}$ position of P^*_{su} along (u,v) are dominated by or are equivalent to a path in P^*_{sv}. The updated final value of $\text{lastProcessedPath}[u,v]$ when the inner loop concludes, is returned by the algorithm s.t. the MDA can pass it to nextQueuePath the next time a queue path for v is searched. This means that in this search, the expansions from paths in P^*_{su} start with the path stored in position $\text{lastProcessedPath}[u,v] = j$ of the list. The paths before the j^{th} position have been proven to be dominated by or equivalent to paths in P^*_{sv}. This proves the statement. $\square$

We can now fully characterize a path's cycle from its first exploration until it is either made permanent or discarded in the MDA. Consider an explored s-v-path $p = q \circ (u,v)$. It becomes an explored path in Line 2 of propagate in the iteration in which the s-u-path q becomes a permanent path. Assume it is the i^{th} permanent path stored in P^*_{su} for some $i \leqslant N_u$. If p does not become a permanent s-v-path in a later iteration, it is discarded after the $\text{lastProcessedPath}[(u,v)]$ index becomes greater than i in Line 6 of nextQueuePath. Of course there might be s-v-paths that are inserted into Q directly after they become explored paths and never leave Q before they are extracted and made permanent. These paths' *life cycle* in the MDA is not influenced by the nextQueuePath subroutine.

Since the outmost loop in nextQueuePath iterates over all incoming arcs of v, we can state the following result.

Theorem 4.3. *nextQueuePath solves* (8).

Proof. After the discussion in the last paragraphs and in particular using Invariant 4.1 it is clear that for every node $u \in \delta^-(v)$ the inner loop of nextQueuePath stops when it finds the $\prec_Q$-smallest s-v-path coming from a permanent s-u-path in P^*_{su} that is neither dominated by nor equivalent to a path in P^*_{sv}.

When nextQueuePath begins, a dummy path p is initialized to NULL and it is assumed to have infinite cost. Fix $u \in \delta^-(v)$. At most one update of p is attempted in Line 10 the first time a new found s-v-path coming from u turns out not to dominated by or equivalent to a path in P^*_{sv}. Then, the inner loop is interrupted by the **break** statement in Line 11. This update is only conducted if the candidate path coming from u is $\prec_Q$-smaller than the current s-v-path p. Hence, the s-v-path p at the end of nextQueuePath is a solution to (8). □

Remark 4.1 (Repeated $\preceq_D$-checks)**.** *Assume a new queue path for a node $v \in V$ is searched in nextQueuePath during the i^{th}, $i \in \mathbb{N}$, iteration of the MDA. Consider an arc $(u, v) \in \delta^-(v)$ and let the* lastProcessedPath$[u, v]$ *index point at path p in P^*_{su} at the end of the search. The next time a new queue path for v is searched because an s-v-path p^* is extracted from Q,* lastProcessedPath$[u, v]$ *is unaltered at the beginning of the search and thus $q = p \circ (u, v)$ is reconsidered. Moreover, between the search in the i^{th} iteration and the current search, the list P^*_{sv} has only been modified by appending p^* to it. Thus, we have*

$$c(P^*_{sv}) \preceq_D c(q) \Leftrightarrow c(p^*) \preceq_D c(q). \tag{9}$$

This means that for every arc $(u, v) \in \delta^-(v)$, the first iteration of the inner loop in nextQueuePath does not need to consider any path in P^*_{sv} other than p^*. Then, the number of comparisons to answer the $\preceq_D$ -check drops from $\mathcal{O}(d|P^*_{sv}|)$ to $\mathcal{O}(d)$. While this complexity drop is not enough to have an impact in the overall asymptotic behavior of the MDA, it enhances the performance in practice noticeably. The technique is easy to explain and implement but harms the readability of the pseudocode which is why we do not include it in nextQueuePath. We refer the reader to our implementation of the MDA in (Maristany de las Casas, 2023a) for the details.

We start our transition towards the study of the output sensitive running time and space consumption bounds of the MDA.

Lemma 4.6. *Consider a node $v \in V$ and a permanent s-u-path p in P^*_{su} for $u \in \delta^-(v)$. The s-v-path $p \circ (u, v)$ is built at most $N_v + 1$ times in all calls to nextQueuePath with v as its input.*

Proof. The node v is the input node for nextQueuePath in Line 10 of the MDA in every iteration of the MDA in which an s-v-path is extracted from Q in Line 7. Every time this happens, the extracted path is appended to P^*_{sv} in Line 8. In the iteration in which the last path in P^*_{sv} is stored, nextQueuePath is called one last time to try to find a new queue path for v. Since no new s-v-paths are added to P^*_{sv}, this search returns NULL. Hence, a new queue path for v calling nextQueuePath is searched $N_v + 1$ times.

In every call if $p \circ (u, v)$ is considered in Line 5 of nextQueuePath, it is considered only in one iteration of the inner loop of nextQueuePath. This iteration either ends with the **continue** statement in Line 8 after which the next iteration of the inner loop begins and a new s-v-path is built or it ends after the **break** statement in Line 11 after which a new iteration of the outer loop of nextQueuePath starts. Hence, $p \circ (u, v)$ is built at most once in every call to nextQueuePath with v as its input. □

An immediate question arising after Lemma 4.6 is if it is possible that all paths in P^*_{su} for $u \in \delta^-(v)$ need to be considered $N_v + 1$ times during the searches for new queue paths for v in nextQueuePath. Luckily that is not the case as we prove in the next two lemmas.

Lemma 4.7. *For every arc $(u, v) \in A$ there holds* lastProcessedPath$[(u, v)] = N_u$ *at the end of the MDA.*

Proof. The index $\text{lastProcessedPath}[(u,v)]$ is initialized to 1. It is increased by one in Line 6 in every new iteration of the inner loop of nextQueuePath. Assume a new queue path for v is searched for the last time during the MDA. This implies that in this search, no new queue path for v is determined.. Otherwise a new queue path for v is inserted into Q and, in a later iteration, also extracted and made permanent. Thus, the update line of p (Line 10) is not reached during this call to nextQueuePath and every iteration of the inner loop ends with the **continue** statement in Line 8. The loop itself ends when $k = N_u$. This proves the statement because in every iteration of the inner loop in nextQueuePath $\text{lastProcessedPath}[(u,v)]$ takes the value of k. □

Lemma 4.8. *For any arc $(u,v) \in A$, at most $N_v + N_u + 1$ dominance or equivalence ($\preceq_D$) checks are conducted in Line 7 of nextQueuePath during all searches for a next queue path for v.*

Proof. A direct consequence of Lemma 4.7 is that every path in P^*_{su} is expanded during the searches for a new queue path for v in the MDA at least once. After every such expansion, a $\preceq_D$-check is conducted in Line 7 of nextQueuePath. This explains the N_u summand in the statement.

The MDA searches for a new queue path for v exactly $N_v + 1$ times and after these searches, $\text{lastProcessedPath}[(u,v)] = N_u$ by Lemma 4.7. This implies that if the kth path in P^*_{su}, $k \in \{1, \dots, N_u\}$, is expanded along (u,v) for the first time during the ith search for a new queue path for v, $i \in \{1, \dots, N_v + 1\}$, the index $\text{lastProcessedPath}[(u,v)]$ can remain at position k during at most $N_v + 1 - i$ calls to nextQueuePath with v as its input node.

All in all, without knowing exactly how often every path in P^*_{su} is expanded along (u,v), we know that the sum of all expansions of paths in P^*_{su} along (u,v) is at most $N_v + 1$. Hence, the total number of $\preceq_D$-checks for an arc (u,v) is at most $N_u + N_v + 1$. □

Dominance checks using $\preceq_D$ are the main driver of comparisons in the asymptotic running time bound of the MDA. Knowing how many of them are conducted for every arc in the graph during calls to nextQueuePath enables us to derive the asymptotic bounds for the MDA in the next section.

4.4 SPACE AND RUNNING TIME COMPLEXITY

We denote the cardinality of the biggest minimal complete set of efficient paths output by the MDA by $N_{\max}$, i.e.,

$$N_{\max} := \max_{v \in V}\{|P^*_{sv}|\} = \max_{v \in V}\{N_v\}.$$

Moreover, we set N to be the total number of efficient paths computed by the MDA, i.e., $N := \sum_{v \in V} N_v$. Since in every iteration of the MDA the path extracted from Q is added to the corresponding set of efficient paths, N coincides with the overall number of iterations in the MDA.

We assume that Q is a *Fibonacci Heap* (Fredman & Tarjan, 1987). In every iteration the extraction of an s-v-path p from Q requires $\mathcal{O}(\log(dn))$ time since the size of the queue is bounded by the number of nodes in the graph. The addition of p at the end of the list P^*_{sv} in Line 8 of the MDA is a $\mathcal{O}(1)$ operation.

Remark 4.2 (Storage of permanent paths using Dimensionality Reduction)**.** *In Section 3.5 we discussed how the dimensionality reduction technique can be used to speedup dominance checks in practice. We further remarked that we use the technique implicitly in our description of label-setting MOSP algorithms. To use it in practice, we need to maintain, for every list P^*_{sv} of permanent paths, an associated dimensionality reduced front $c_{dr}(P^*_{sv}) \subset \mathbb{R}^{d-1}$. Maintaining this list correctly requires the use of merge operations. Thus, when an s-v-path p is extracted from Q in the MDA, we update the list $c_{dr}(P^*_{sv})$ and this takes $\mathcal{O}(d|P_{sv}|^*) \subset \mathcal{O}(dN_{\max})$ according to Proposition 3.1. All in all, for the extraction and storage of permanent paths, the MDA requires $\mathcal{O}(N(\log dn + dN_{\max}))$ if dimensionality reduction is used. Since $N \leqslant nN_{\max}$ we get an asymptotic running time of*

$$\mathcal{O}\left(N\log(dn) + dnN^2_{\max}\right) \tag{10}$$

for the extraction and storage of permanent paths in the MDA if dimensionality reduction is used. We will see later that asymptotically, the second summand can be neglected.

If dimensionality reduction is not used, the second summand in (10) is neglected. In what follows, we bound, for every arc, the effort made during the calls of the MDA to its two subroutines propagate and nextQueuePath.

COMPLEXITY OF CALLS TO PROPAGATE Let $(v, w) \in A$ be an arc. For every s-v-path p that is extracted from Q, the path $q := p \circ (v, w)$ is build in Line 2 of propagate. For every such path q, this is the first time it is analyzed by the MDA. The dominance check in Line 3 of propagate requires $\mathcal{O}(dN_w)$ comparisons. This linear time check w.r.t. the size of P^*_{sw} dominates the remaining operations in propagate. If q is ignored because it is dominated by or equivalent to a path in P^*_{sw}, nothing else happens; otherwise q is either inserted into Q directly (Line 8) or it updates an existing s-w-path in Q (Line 5). Using a Fibonacci Heap, the insertion takes $\mathcal{O}(\log(dn))$ time and the update $\Theta(d)$ time. Hence if the MDA stores N_v permanent and efficient s-v-paths, the overall effort for the propagations of these paths along the arc (v, w) is $\mathcal{O}(dN_vN_w) \in \mathcal{O}\left(dN^2_{\max}\right)$ since N_v and N_w are bounded by $N_{\max}$. If we consider all arcs in the input graph G, we obtain that the expansions of extracted paths in the MDA take $\mathcal{O}\left(dmN^2_{\max}\right)$ time.

COMPLEXITY OF CALLS TO NEXTQUEUEPATH The number of $\preceq_D$ -checks triggered in nextQueuePath for an arc $(u, v) \in A$ is in $\mathcal{O}(N_{\max})$ (Lemma 4.8). Every $\preceq_D$-check uses $\mathcal{O}(dN_{\max})$ comparisons. Thus, as in propagate, the calls to nextQueuePath for an arc (u, v) use $\mathcal{O}\left(dN^2_{\max}\right)$ time. Summing over all arcs, we get an overall complexity of $\mathcal{O}\left(dmN^2_{\max}\right)$ for the searches for new queue paths after the extraction of a path from the queue Q of the MDA.

In the next theorem, recall that extracting and making paths permanent is bounded in (10).

Theorem 4.4 (Time Complexity of the MDA)**.** *The MDA using*

- *a Fibonacci Heap for the priority queue* Q*,*
- *vectors for the sets* P^*_{sv} *of permanent* s*-*v*-paths for all* $v \in V$*,*
- *and the flags* $\text{lastProcessedPath}[a]$ *for every arc* $a \in A$ *to efficiently solve* (8) *using nextQueuePath.*

runs in

$$\mathcal{O}\left(N \log dn + dnN^2_{\max} + dmN^2_{\max} + dmN^2_{\max}\right) = \mathcal{O}\left(N \log(dn) + dmN^2_{\max}\right) \tag{11}$$

using the common connectivity assumption of G *that allows us to assume* $n < m$*.*

Note that without using dimensionality reduction we would get the right hand side of (11) directly (see also Remark 4.2). Thus, the use of dimensionality reduction is asymptotically negligible, even though if requires a merge operation in every iteration to maintain the dimensionality reduce fronts.

The bound in the last theorem is driven by the complexity of the $\preceq_D$-checks performed during the MDA. As explained in Section 3.5 in biobjective scenarios these checks can be performed in $\Theta(1)$ using the dimensionality reduction technique. Thus, we derive the following improved bound for the MDA for BOSP instances.

Theorem 4.5 ((Sedeño-Noda & Colebrook, 2019, Theorem 3))**.** *The MDA using dimensionality reduction for the dominance checks solves biobjective* ($d = 2$) *MOSP instances in* $\mathcal{O}\left(N \log n + N_{\max} m\right)$*.*

The space consumption of the MDA is easier to derive. The graph and the arc cost vectors are stored using $\mathcal{O}\left(n + dm\right)$ space. Additionally, the algorithm only uses the priority queue Q whose size is bounded by n and the lists P^*_{sv} of permanent s-v-paths for every $v \in V$. The overall size of these lists is N. Moreover, for every arc $a \in A$, the index $\text{lastProcessedPath}[a]$ is stored. Hence, we immediately get the following result if we assume that paths are stored as labels (cf. Section 3.3) that require $\mathcal{O}\left(d\right)$ memory each.

Theorem 4.6 (Space Complexity of the MDA)**.** *The space required by the MDA using the same data structures as described in Theorem 4.4 is*

$$\mathcal{O}\left(dN + n + dm\right). \tag{12}$$

4.5 CONCLUSION

Why is the MDA novel? Are its running time and space complexity bounds an improvement w.r.t. the state of the art? How does it perform in practice? These questions need to be answered before it becomes clear why the MDA is the main contribution in this thesis. In the next chapter, we introduce an improved version of the classical label-setting MOSP algorithm by E. Q. V. Martins (1984). Afterwards, in Chapter 6, we compare both algorithms.

5 MARTINS'S ALGORITHM

The classical label-setting MOSP algorithm is due to E. Q. V. Martins (1984). Algorithms for One-to-One MOSP like the NAMOA* algorithm from Pulido et al. (2015, 2014) or to approximate MOSP solutions like in (Breugem et al., 2017) are based on Martins's algorithm. Despite its undeniable relevance and its extended use in publications, we could not find a high quality, modern, and openly available implementation of the algorithm. Similarly noticeable, to the best of our knowledge, only the work by Breugem et al. (2017) contains an asymptotic running time analysis of the algorithm. However, their derived bound is too high. This can be a consequence of the fact that the original publication (E. Q. V. Martins, 1984) leaves some room for interpretation regarding the use of data structures and in particular w.r.t. the handling of explored paths in the algorithm's queue.

Martins's algorithm heavily relies on merge operations in every iteration. In this chapter, we present a version of the algorithm that achieves the best possible asymptotic running time bound staying close to the original description. In other words, we stretch the interpretation of the high level pseudocode given in (E. Q. V. Martins, 1984) to design a version of the algorithm s.t. its comparison with the MDA boils down to the comparison of the usage of the nextQueuePath and the merge subroutines. In (Maristany de las Casas, Sedeño-Noda, & Borndörfer, 2021), the publication in which we introduced the MDA, we compared the algorithm against a state of the art version of Martins's algorithm from the literature published in (Demeyer et al., 2013). In this paper, the speedup achieved using the MDA was bigger than the one presented in Chapter 6. However, the inefficiencies in the chosen data structures to handle explored paths are obvious and not demanded in Martins' original publication. We thus, designed a new version of Martins's algorithm and introduce it in this chapter (Section 5.1). We prove its correctness (Section 5.2) and derive its asymptotic bounds (Section 5.3).

5.1 DESCRIPTION OF THE ALGORITHM

The pseudocode of our version of the label-setting One-to-All MOSP algorithm by Martins (E. Q. V. Martins, 1984) is in Algorithm 6. We refer to it using the Martins link during the remainder thesis. We assume a One-to-All MOSP instance $(G = (V, A), s, d, c)$ is given throughout the chapter.

DESCRIPTION The algorithm uses lists P^*_{sv} of permanent paths for every $v \in V$. For a fixed node, the corresponding list, at the end of the algorithm, contains a minimal complete set of efficient s-v-paths. For every node a list E_{sv} of explored paths is maintained. Additionally a priority queue Q sorted w.r.t. a total order $\prec_Q$ that is compatible with the dominance order, stores at most one s-v-path for every $v \in V$. All data structures are initially empty (Line 1). In every iteration an explored path is extracted from Q (Line 5). It is then guaranteed to be an efficient s-v-path for its end node $v \in V$. The remaining explored s-v-paths are stored in the list E_{sv}. No two paths in the list dominate each other or are cost-equivalent. Moreover, no path in the list is dominated by or equivalent to the paths in P^*_{sv}. The paths in the list are sorted in ascending order w.r.t. $\prec_Q$ and thus, when an s-v-path p is extracted from Q, the first element in E_{sv} if it exists, is inserted into Q as v's new queue path (Line 7). In the same iteration, p is expanded along the outgoing arcs of v and new explored paths $q = p \circ (v, w)$ are constructed (Line 9). If q is dominated by or equivalent to a path in P^*_{sw} it is immediately discarded (Line 10). Otherwise, using merge operations, q is merged into the list E_{sw} (Line 12). If q is not only determined to be a relevant explored path but also the $\prec_Q$-smallest one, it replaces w's queue path q' in Q (Line 14) and q', that was previously the smallest explored s-w-path, is added at the beginning of E_{sw} (Line 15). If no queue path for w exists when q is built, there are no explored s-w-paths and thus, q is directly inserted into Q (Line 17). The algorithm ends when Q is empty at the beginning of an iteration. It returns all sets P^*_{sv}.

Algorithm 6: Martins' One-to-All MOSP algorithm

```
Input  : d-dimensional One-to-All MOSP instance $(G, s, d, c)$.
Output: Minimal complete set $P_{sv^*}$ of minimal $s$-$v$-paths for every
        $v \in V$.
1  Prio. queue $Q$ and explored paths $E_{sv}$ and permanent paths $P^*_{sv}$
     $\forall v \in V$ empty;
2  Initial dummy $s$-$s$-path $p_{init}$ with $c(p_{init}) = 0$;
3  $Q$.insert$(p_{init})$;
4  while $Q \neq \emptyset$ do
5      $s$-$v$-path $p \leftarrow Q$.extractMin$()$;
6      $P^*_{sv} \leftarrow$ Append $p$ to $P^*_{sv}$;
7      if $E_{sv} \neq \emptyset$ then Insert $\prec_Q$-smallest $s$-$w$-path from $E_{sw}$ into $Q$;
8      foreach $(v, w) \in \delta^+(v)$ do
9          New explored $s$-$w$-path $q \leftarrow p \circ (v, w)$;
10         if $c(P^*_{sw}) \preceq_D c(q)$ then continue ;
11         if $Q$ contains an $s$-$w$-path $q'$ then
12             Use merge: determine if $c(E_{sw} \cup \{q'\}) \preceq_D c(q)$, discard
                 paths in $E_{sw} \cup \{q'\}$ dominated by $q$, and (possibly) insert
                 $q$ into $E_{sw}$;
13             if $q$ is the new $\prec_Q$-smallest explored $s$-$w$-path then
14                 Replace $q'$ with $q$ in $Q$;
15                 if not $c(q) \preceq_D c(q')$ then $E_{sw} \leftarrow$ Prepend $q$ to $E_{sw}$ ;
16         else
17             Insert $q$ into $Q$;
18 return $\cup_{v \in V} P^*_{sv}$.
```

The techniques to prove the correctness of Martins's algorithm are similar to the ones used in the MDA. Moreover, we assume the reader has by now a strong intuition of how paths are handled in a label-setting MOSP algorithm using compatible orders. Thus, the following properties are stated without a detailed proof. In Section 5.2 we sketch the remaining correctness proof. The original version of the algorithm does not use the lists E_{sw} for $w \in V$, the detailed correctness proof can be read in (E. Q. V. Martins, 1984).

Invariant 5.1 (Martins' Explored Paths)**.** *At the end of any iteration in Martins's algorithm, the following conditions hold for any node $w \in V$.*

1. *There is at most one s-w-path in Q. It is smaller w.r.t. $\prec_Q$ than any s-w-path in E_{sw} and does not dominate any s-w-path in E_{sw}.*
2. *Paths in E_{sw} are not dominated by any path in P^*_{sw} and no two paths in E_{sw} dominate each other.*
3. *If there is no s-w-path in Q, E_{sw} is empty.*

5.2 CORRECTNESS

The following statement is trivial. The proof can be conducted inductively using that the path from s to itself with costs $0 \in \mathbb{R}^d$ is a subpath of every path extracted from Q during Martins's algorithm.

Proposition 5.1. *Assume that Martins's algorithm extracts an s-v-path p from Q in iteration $k \in \mathbb{N}$ of its main loop. Then, at the beginning of every iteration $l < k$ of the algorithm, there is one node $u \in p$ for which the s-u-subpath $p^{s \to u}$ of p is explored, i.e., $p^{s \to u}$ is in Q or in E_{su}.*

The next lemma is a direct consequence of Proposition 5.1.

Lemma 5.1. *If a path p is extracted in the k^{th} iteration of Martins's algorithm, then at the end of every previous iteration there exists a node $u \in p$ for which an s-u-path is in Q.*

Proof. By Invariant 5.1 if for all nodes u along p there is no s-u-path in Q, the corresponding lists E_{su} are empty. In this situation, no subpath of p would be an explored path, which contradicts Proposition 5.1. □

We can now formulate a statement that is equivalent to Lemma 4.4 in the correctness proof of the MDA.

Lemma 5.2. *Consider two paths p and q in G if p is extracted from Q before q, then $c(p) \preceq_Q c(q)$.*

Proof. Assume p is extracted in the k^{th} iteration of Martins's algorithm. If p and q are simultaneously in Q at the beginning of this iteration, then the statement is trivially true because only the top element of Q is extracted and paths therein are sorted in nondecreasing order w.r.t. $\prec_Q$. Assume q is not in Q when p is extracted. By Proposition 5.1 we know that for a node $u \in q$, the

s-u-subpath $q^{s \to u}$ of q is either in Q or in E_{su}. Recall that $c(q_{su}) \preceq_Q c(q)$ because arc costs are nonnegative. If $q^{s \to u}$ is in Q, we have

$$c(p) \preceq_Q c(q^{s \to u}) \preceq_Q c(q)$$

because otherwise p would not be extracted in the current iteration. If $q^{s \to u}$ is not in Q, it is in E_{su}. The logical negation of the third point in Invariant 5.1 guarantees that in this case, there exists an s-u-path q' in Q. From the first point in Invariant 5.1, we conclude that $c(q') \prec c(q_{su})$. Then, we have

$$c(p) \preceq_Q c(q') \prec_Q c(q^{s \to u}) \preceq_Q c(q)$$

because, again, p would not be extracted in the current iteration otherwise. $\square$

Finally, Theorem 5.1 states that Martins's algorithm outputs minimal complete sets of efficient paths. Proving Theorem 5.1 is equivalent to proving Theorem 4.2 for the MDA. Also this remaining proof does no longer depend on our new lists E_{sv} of explored paths and thus it coincides with the proof in (E. Q. V. Martins, 1984, Assumption 2, Assumption 3, Lemma 4).

Theorem 5.1 (Correctness of Martins's Algorithm). *Let $C_{sw} \subset \mathbb{R}^d$ be the set of nondominated cost vectors induced by all efficient s-w-paths in G. For every $c \in C_{sw}$ Algorithm 6 outputs one s-w-path p s.t. $c(p) = c$.*

5.3 COMPLEXITY

An asymptotic analysis of Martins's algorithm is rarely done in the literature and our version of the algorithm has the lowest output sensitive running time bound that we could find in previous publications.

We prove that Martins's algorithm is output sensitive. As always, given a One-to-All MOSP instance $\mathcal{I} = (D, s, d, c)$, we denote the number of nodes and arcs in G by n and m, respectively. Moreover, we refer to the cardinality of the largest output set by $N_{\max}$, i.e., $N_{\max} := \max_{v \in V} |P^*_{sv}|$ and to the overall number of output paths by N, i.e., $N := \sum_{v \in V} |P^*_{sv}|$. Since in the lists P^*_{sv} of permanent paths we only need to append paths in Line 6 we assume them to be single linked lists or vectors. The lists of explored paths are doubly linked lists to be able to remove and insert elements in any position during merge operations in constant time. The priority queue Q is a *Fibonacci Heap* (Fredman & Tarjan, 1987).

The running time of Martins's algorithm is dominated by the running time of the merge operations. In Proposition 3.1 we already derived that this operation takes linear time w.r.t. the cardinality of the set into which the new element is being merged. In our case, these sets are the sets/lists E_{sv} of explored paths. Explored paths are always generated from permanent paths that are extracted from Q. The number of permanent paths for any node is bounded by $N_{\max}$. In a worst case scenario a node $v \in V$ can have $(n-1)$ predecessor nodes, i.e., $|\delta^-(v)| = n-1$. Thus, the cardinality of E_{sv} is bounded by $\mathcal{O}(nN_{\max})$.

Proposition 5.2. *Every merge operation in Line 12 of Martins's algorithm takes $\mathcal{O}(dnN_{\max})$ in MOSP instances with $d \geqslant 3$ arc cost components.*

The Two-Dimensional Merge Dilemma

The running time bound of the MDA is lower for biobjective instances (cf. Theorem 4.5). The algorithm uses merge operations to maintain the dimensionality reduced fronts but in the bidimensional scenario these fronts consist only of a single number and thus, merge operations run in $\Theta(1)$ time. In the biobjective scenario, Martins's algorithm uses merge operations on the lists of explored paths that can store $nN_{\max}$ elements also in the $d = 2$ scenario. Thus, we need to answer the question: Can we reduce the asymptotic running time bond for Martins's algorithm for BOSP instances?

Let p be an s-v-path for any $v \in V$ and E_{sv} the list of explored paths in Martins's algorithm during the solution process of a BOSP instance. If E_{sv} is a data structure whose elements can be accessed in constant time using their index in E_{sv}, we can use two binary searches to find out if E_{sv} contains a path that dominates p or is equivalent to p, to determine where p needs to be inserted to keep E_{sv} sorted, and to find out which paths in E_{sv} are dominated by p. This is only a *pseudo-merge* because it does not actually delete the dominated paths from E_{sv}.

Lemma 5.3. *The pseudo-merge operation described in the last paragraph requires* $\mathcal{O}(\log(nN_{\max}))$ *comparisons in MOSP instances with* $d = 2$ *arc cost components if* E_{sv} *is sorted in lexicographically ascending order and implemented as a data structure whose elements can be accessed in constant time using their indices.*

Proof. By Invariant 5.1 no two paths in E_{sv} dominate each other or are cost-equivalent. Then, the c_1 cost components of the paths in E_{sv} are sorted in ascending order and the c_2 cost components are implicitly sorted in descending order. Let p be a new explored s-v-path and suppose we merge p into E_{sv}. A first binary search among the first cost components of the paths in E_{sv} finds the tentative insertion position of p. I.e., $c(q) \preceq_{lex} c(p)$ for every path q that is in E_{sv} before the found insertion position. Let q be the path immediately before the found position if it exists. We then have $c_1(q) \leqslant c_1(p)$ and if $c_2(q) \leqslant c_2(p)$ we discard p because it is dominated or equivalent to q. Otherwise if $c_2(q) > c_2(p)$ no path q' that is stored in E_{sv} before q dominates p because the sorting of paths in E_{sv} implies $c_2(q') > c_2(q)$.

Similarly, a second binary search is used to check if p dominates paths in E_{sv}. Such paths must come after p's insertion position. Let q be a path checked during the binary search. We have $c_1(p) < c_1(q)$. If $c_2(p) < c_2(q)$, p dominates every path q' between p and q in E_{sv} because $c_2(q') > c_2(q)$ and we can *mark these paths to be removed* from E_{sv} and continue the binary search on the paths stored after q in E_{sv}. Otherwise if $c_2(p) > c_2(q)$, p does not dominate q and also not any path q' that comes after q in E_{sv} because in this case $c_2(q) > c_2(q')$. Thus, we can continue the binary search on the paths between p and q in E_{sv} only. □

The emphasized sentence in the proof is the dilemma that motives this subsection's title. The number of paths that might need to be removed is $|E_{sv}| \leqslant nN_{\max}$. During the second binary search used in the proof, we can mark these paths as dominated in constant time using the indices of the elements. However, deleting or freeing the labels encoding the dominated paths while maintaining the properties of E_{sv} for future merge operations requires linear time if E_{sv} is a vector or a list (elements not accessible in constant time using their indices). We are not aware of a different data structure that allows the bulk deletion of elements in $\mathcal{O}(1)$ time and does not have other negative consequences on the running time of merge operations. If this data structure exists, merge operations can be done in $\mathcal{O}(\log(nN_{\max}))$ when solving BOSP instances. Otherwise, they remain $\mathcal{O}(nN_{\max})$ operations.

Asymptotic Bounds

Let $T_{\preceq_D}$ denote the running time bound for the $\preceq_D$-checks in Martins's algorithm. Similarly, T_{merge} is the running time of the merge operations in Line 12 of Martins's algorithm. For $d \geqslant 3$, we have $T_{\text{merge}} \in \mathcal{O}(dnN_{\max})$. For $d = 2$ we face the dilemma explained in the last subsection.

Theorem 5.2 (Running Time of Martins's algorithm)**.** *Martins's algorithm runs in* $\mathcal{O}\left(N\log(dn) + mN_{\max}(T_{\preceq_D} + T_{merge})\right)$.

Proof. The algorithm does N iterations because every extracted path is efficient and made permanent. Since the queue contains at most n elements, the extraction of a minimal element is done in $\mathcal{O}(\log(dn))$. We know from the complexity analysis of the MDA that the complexity for maintaining dimensionality reduced fronts if the technique is used, can be asymptotically neglected. Thus, we obtain the first summand for the extraction and storage of permanent paths.

The generation of a new queue path in Line 7 is a constant time operation because we maintain the lists E_{sv} without dominated elements and sorted (cf. Invariant 5.1). The insertion of the new path into Q can be asymptotically neglected.

The expansions of a permanent path along the outgoing arcs of its end node is done as in propagate in the MDA but additionally using the merge operation. For an arc $(v, w) \in A$, the $\mathcal{O}(N_{\max})$ permanent paths in P^*_{sv} are expanded along the arc and two tasks are performed: a $\preceq_D$ check (Line 10) that takes $\mathcal{O}(T_{\preceq_D})$ comparisons and a merge into E_{sw} (Line 12) in $\mathcal{O}\left(T_{\text{merge}}\right)$ time. The remaining operations can be asymptotically neglected. Summing over all arcs, we obtain a running time bound of

$$\mathcal{O}\left(m(N_{\max}(dN_{\max} + T_{\text{merge}})\right)$$

for the paths expansions. All in all, the running time bound for Martins's algorithm is

$$\mathcal{O}\left(N\log(dn) + mN_{\max}(T_{\preceq_D} + T_{\text{merge}})\right). \tag{13}$$

□

The space complexity of the Martins's algorithm is easy to derive. The graph and the d-dimensional arc costs are stored in $\mathcal{O}(n + dm)$ space. If paths are stored as labels, the storage of the permanent paths requires $\mathcal{O}(dN)$ memory. Additionally, labels corresponding to explored paths are stored explicitly in the E_{sv} lists. Each of these lists contains $\mathcal{O}(nN_{\max})$ labels. Since $N \leqslant nN_{\max}$ we get the following result.

Proposition 5.3. *The memory consumption bound of Martins's algorithm is*

$$\mathcal{O}(dnN_{\max} + n + dm). \tag{14}$$

	Running Time $\mathcal{O}(\cdot)$		Memory $\mathcal{O}(\cdot)$
	$d = 2$	$d \geqslant 3$	
MDA	$T_{extr} + mN_{max}$	$T_{extr} + dmN_{max}^2$	$dN + dm + n$
Martins	$T_{extr} + mN_{max}nN_{max}$	$T_{extr} + dmnN_{max}^2$	$dnN_{max} + dm + n$

Table 1: Asymptotic comparison of the MDA and Martins's algorithm. For space reasons we use $T_{extr} = N\log(dn)$. The blue term nN_{max} can be replaced by $\log(nN_{max})$ if a suitable data structure to speedup merge operations in biobjective scenarios exists.

Note that (14) hides a $N + nN_{max}$ term for the storage of permanent and explored paths simultaneously. In practice having to store both sets of paths is more challenging than the direct comparison of (14) with the MDA's space consumption bound (12) indicates.

After the asymptotic analysis of Martins's algorithm and using Proposition 5.2 and Lemma 5.3 to specify T_{merge} in (13) we can fill Table 1 to the asymptotic behavior of the MDA and our version of Martins's algorithm.

5.4 CONCLUSION

We have designed a version of Martins's algorithm that splits the storage of explored paths into the algorithm's priority queue and a list of explored s-v-paths for every node v in the input digraph. By doing so, we circumvent the issue of having to store all explored s-v-paths in the priority queue, which demands some type of hashing to access a node's explored paths fast during merge operations and the deletion of random elements from the priority queue. The result is an improved running time bound for Martins's algorithm. Still, the bound is worse than the one derived for the MDA (cf. Table 1). This concludes our theoretical analysis of label-setting MOSP algorithms and we can finally benchmark them on MOSP instances commonly used in the literature.

6 EXPERIMENTS: MARTINS'S ALGORITHM VS. MDA

We briefly summarize why we benchmark our version of Martins's algorithm against the MDA.

Originally, the MDA was published in (Maristany de las Casas, Sedeño-Noda, & Borndörfer, 2021) and we compared it to an improved version of Martins's algorithm presented in (Demeyer et al., 2013). When we implemented our version of Martins's algorithm as described in Chapter 5 we noticed that our version is up to an order of magnitude faster. The main reason is that the version by Demeyer et al. (2013), as described in their pseudocode, seems to store explored paths in the priority queue all together. This forces the use of hashes to enable faster merge operations and to delete elements stored in random positions from the queue. Apart from having an asymptotic running time complexity that is worse than the one derived in Theorem 5.2, in practice this bookkeeping has a negative impact on the algorithm's performance. For these reasons, we decided to repeat the experiments from (Maristany de las Casas, Sedeño-Noda, & Borndörfer, 2021) with our most modern implementation of the MDA and with our implementation of Martins's algorithm as described in Chapter 5.

Both algorithms are implemented to share as much code as possible (cf. Section 6.2). In Section 6.1 we explain when to expect that Martins's algorithm solves MOSP instances faster than the MDA. The arguments in this section help interpreting the results later in this chapter. In Section 6.2 we discuss further implementation details. In Section 6.2.1 we specify the used instances and how they were generated. Finally, in Section 6.3 we report and analyze the results obtained from our experiments.

6.1 STRENGTHS AND WEAKNESSES IN PRACTICE

In this thesis, the main difference between the MDA and Martins's algorithm is the use of the nextQueuePath routine to re-generate explored paths versus the use of merge operations to maintain lists of explored s-v-paths in which no two paths dominate each other. In both algorithms, we sort explored paths lexicographically and use the dimensionality reduction technique (Section 3.5).

A relevant subset of paths is built by those paths that are inserted into the algorithms' queue right after they are explored in Line 2 of propagate during the MDA or in Line 9 of Martins's algorithm and that are never replaced using a decrease key operation before being extracted from Q and made permanent. For a node $v \in V$, an s-v-path p, and an arc $(u,v) \in \delta^-(v)$ let $q = p \circ (u,v)$ be such a path. Then, Martins's algorithm performs the check $c(P^*_{sv}) \preceq_D c(q)$ once when q becomes an explored path and also performs the corresponding merge operation in this iteration. No more comparisons are required before making q permanent. In the MDA the check $c(P^*_{sv}) \preceq_D c(q)$ is also performed when q becomes an explored path during a call to propagate. However, p is a permanent s-u-path and is thus stored in P^*_{su}. Thus, in a later iteration of the MDA, the search for a next queue path for v is triggered and during the involved call to nextQueuePath, the $\mathrm{lastProcessedPath}[u,v]$ index will point to the entry in P^*_{su} containing p. Then, q is rebuilt and $c(P^*_{sv}) \preceq_D c(q)$ is repeated.

As a consequence of the above paragraph, we can conclude than whenever the merge operations during Martins's algorithm are fast and relatively many paths are made permanent without the need to be replaced in Q, the need to repeat $\preceq_D$ -checks in the MDA causes Martins's algorithm to be faster. Our experiments indeed show that instances with a small overall number N of permanent paths stick to this logic.

It is to be expected that the MDA outperforms Martins's algorithm when the management of explored paths becomes more tedious. The asymptotic behavior of both algorithms (cf. Table 1) seconds this claim. merge operations, also in practice, are time consuming and cannot keep up with the straightforward iteration required to answer (repeated) $\preceq_D$-checks.

6.2 IMPLEMENTATION DETAILS

Both algorithms are implemented in C++ and we use the gcc compiler version 7.5 to build the binaries setting the compiler optimization level to O3. To describe an s-v-path p, we use a label (cf. Section 3.3) that is a tuple $(v, c(p), (u,v), \mathrm{pred}(p))$. In the label, (u,v) is the last arc of p and $\mathrm{pred}(p)$ is a pointer to the label encoding the (permanent) s-u-subpath $p^{s\to u}$ of p. The implementations are accessible in (Maristany de las Casas, 2023a). The code can be compiled to solve instances with any number of arc cost functions.

Remark 6.1 (Explored and permanent labels in Martins's algorithm)**.** *Consider implementations of Martins's algorithm and of the MDA both using dimensionality reduction. I.e., alongside with the lists P^*_{sv} the algorithms maintain the dimensionality reduced fronts $c_{dr}(P^*_{sv})$ to speedup $\preceq_D$ -checks (cf. Section 3.5). For any $v \in V$ assume p is a permanent s-v-path in both Martins's algorithm and the MDA. In the MDA, p needs to be stored in P^*_{sv} using a label that includes p's costs. Otherwise, during calls to nextQueuePath we cannot recalculate the costs of the explored path $p \circ (v,w)$ efficiently. In Martins's algorithm if p is made permanent, it is not reconsidered until it is rebuilt recursively using Algorithm 1 at the end of the algorithm for output purposes. Since Algorithm 1 iterates over all arcs in p to rebuild it, we can sum up the costs of the arcs to calculate p's cost. Hence, in Martins's algorithm, we can use different labels for explored paths and for permanent paths. Namely, permanent paths can be stored using a label that does not include the paths' cost vector. This saves memory in practice and avoids reallocation of large chunks of memory when the P^*_{sv} vectors grow. This trick was first used or at least explained in a publication related to label-setting One-to-One BOSP problems by Ahmadi et al. (2021).*

Data Structures

Labels in both algorithms are allocated in a *memory pool*, large blocks (arrays) of memory linked to each other. This approach helps avoiding the fragmentation of memory. If a label is discarded, it is *freed* in the pool meaning that when the algorithm requests a new label to encode an explored path, the discarded label is made available and its information is overwritten.

In both algorithms we use a binary heap as the priority queue Q of explored paths. It contains at most n elements, one s-v-path for every $v \in V$. Both algorithms use a hash to access a node's queue path quickly. The lists P^*_{sv} of permanent s-v-paths/labels are implemented as vectors. We only need to append elements to these lists in both algorithms. In our implementation of Martins's algorithm, the lists E_{sv} of explored s-v-paths are doubly linked lists. During merge operations we need to be able to insert at and delete from any position in the list. The indices lastProcessedPath for every arc $a \in A$ are needed in the MDA and we store them as an array indexed by the ids of the arcs.

Finally, we maintain the dimensionality reduced fronts (cf. Section 3.5) in both algorithms to speed up dominance tests. They are maintained as doubly linked lists to enable fast insertion and removal from arbitrary positions.

6.2.1 Instance description

The instances from Hansen (1980) used to prove the intractability of MOSP problems are excellent benchmark instances. Since all paths are efficient and no two paths with the same end nodes are cost-equivalent, all dominance checks performed by any label-setting algorithm require the same amount of comparisons. If storage of the graph and the permanent lists coincide like in our implementations, the differences in running times boil down to the comparison of the desired subroutines: nextQueuePath for the MDA versus merge for Martins's algorithm. Also, these instances do not distinguish between a One-to-All and a One-to-One setting since all paths are efficient and thus cannot be pruned. We refer to these instances as *EXP instances* in this section and report the results in Section 6.3.1. We choose the name EXP because of the exponential output size. Note that these graphs have bidimensional arc costs originally. We add a third cost component to every arc that is always equal to one. This does not change the fact that every path is efficient.

Besides the instances from Hansen (1980), we choose the same instances as in (Maristany de las Casas, Sedeño-Noda, & Borndörfer, 2021) to compare Martins's algorithm and the MDA on different One-to-One MOSP instances with $d = 3$. Our intention is to observe the behavior of the algorithms on large scale and practically relevant MOSP instances for which data is available in the literature. In Chapter 9 we design a new variant of the MDA tuned for practical performance on One-to-One MOSP instances. In this section we stick to the MDA as described so far because it achieves the best asymptotic memory consumption bound. Recall that $t \in V$ is the target node in One-to-One MOSP instances. The only modification to the MDA and to Martins's algorithm is that whenever we check $c(P^*_{sv}) \preceq_D c(p)$ for an s-v-path p, $v \in V$, we also check $c(P^*_{st}) \preceq_D c(p)$. If this last check is answered positively, p can be discarded because due to the nonnegative arc costs any expansion of p until t does not result in an efficient s-t-path. This technique is called *target pruning* in the literature. We discuss it in more detail in Section 9.4.

GRID GRAPHS We consider a directed 100×100 grid graph. Arcs between neighboring nodes exist in both directions. This results in a graph with 10000 nodes and 39600 arcs. Every arc (u, v) has 3-dimensional costs and $c((u, v)) = c((v, u))$. The costs were generated uniformly at random for each component separately. The values range between 1 and 10. We use 10 different 3-dimensional cost functions. These instances were already used and described in (Pulido et al., 2014). We assign an ID to every node. The IDs start at 0 at the lower left node of the grid and increase first vertically (the node on the upper left corner has id 99) and then horizontally (the node to the right of node 0 has id 100). For every pair consisting of the grid graph and a cost function, we create 100 (s, t) pairs randomly. Since we consider 10 cost functions, we get 1000 One-to-One MOSP instances which we refer to as *Grid Instances*.

NETMAKER INSTANCES NetMaker graphs are synthetic directed graphs with 5000 to 30000 nodes and 29591 to 688398 arcs. These graphs have been used in multiple publications (Maristany de las Casas, Sedeño-Noda, & Borndörfer, 2021; Raith & Ehrgott, 2009; Skriver & Andersen, 2000; Raith et al., 2018). In every such graph, all nodes are connected via a Hamiltonian cycle to ensure connectivity. Then, arcs between the nodes along the cycle are added randomly. A parameter bounds the number of nodes each of the additional arcs can skip over. The arcs' costs are 3-dimensional and each cost component lies between 1 and 1000. For any arc there is a cost component between 1 and 333, a cost component between 334 and 666, and a cost component between 667 and 1000. Which cost component lies in which interval and the actual value of the cost component was randomly set when the graphs were created. We partition the NetMaker graphs into groups called *netM*-n. Each group is defined by the number $n10^3$ of nodes of the graphs in it (i.e., the netM-5 group contains graphs with 5000 nodes) and contains 10 to 12 graphs with varying number of arcs. We use the same s-t-pairs used in (Maristany de las Casas, Sedeño-Noda, & Borndörfer, 2021): for each graph, 20 randomly generated s-t-pairs are considered. Thus, in every netM-n group 200-240 instances are considered.

ROAD NETWORKS We use the road networks available from the *9th DIMACS Implementation Challenge on Shortest Paths* (Demetrescu et al., 2009). The available arc cost functions are the arcs' distance and the arcs' travel time. Additionally, we added a third cost component to every arc that is always 1, i.e., we aim to minimize the number of arcs along a path as a third cost component. For every considered graph, we selected 100 random s-t-pairs to generate our instances. The sizes of the graphs are shown in Table 2.

Road Network	Nodes	Arcs
NY	264,346	733,846
BAY	321,270	800,172
COL	435,666	1,057,066
FLA	1,070,376	2,712,798
NE	1,524,453	3,897,636
LKS	2,758,119	6,885,658
E	3,598,623	8,778,114
W	6,262,104	15,248,146
CTR	14,081,816	34,292,496

Table 2: Size of the used road networks. In every network, we consider 100 random s-t-pairs.

For every instance type, the chosen s-t-pairs on every graph are accessible in text files using the .inst file type and stored in directory instances in the repository (Maristany de las Casas, 2023a).

6.3 RESULTS

The experiments were run on a computer with an Intel Xeon-Gold-6342 @ 2.80GHz processor. For every instance, the available memory was 64GB and we set the time limit to 2h. In Section 6.3.2-Section 6.3.4 we analyze the results for every graph type separately. The reported averages are geometric means and time is always measured in seconds. The collected statistics for every single instance are available in the .csv files in the results directory in (Maristany de las Casas, 2023a). The repository also contains the evaluation scripts used to build the tables and the plots in this section from the collected data. Speedups are calculated by dividing Martins's algorithm running time by the MDA's running time. Hence, whenever the reported speedups are above 1, the MDA is faster than Martins's algorithm.

Remark 6.2 (Unsolved instances)**.** *We consider only instances that are solved by at least one of the algorithms. Some instances are solved only by the MDA. We assume that Martins's algorithm needs* 2h *to solve such instances. Then, we can report time and speedup averages in every row of the tables in this section. However, the reported average number of iterations* (N) *and average number of efficient* s-t-*paths* (N_t) *are not comparable on instance groups in which Martins's algorithm solves less instances than the MDA. These averages are built considering only the instances solved by each of the algorithms.*

6.3.1 Exponential Instances

In Table 3 we collect the results obtained from the EXP instances. The table contains one row per instance. The graph EXPn represents the graph from (Hansen, 1980) with n nodes. See also Figure 2 to recall how the graphs are constructed. We do not include the graphs with $\leqslant 19$ nodes in the table because both algorithms solve them in less than ten milliseconds. Figure 4

			time		
Graph	N	N_t	MDA	Martins	Speedup
EXP15	382	128	0.0002	0.0002	1.00
EXP17	766	256	0.0004	0.0005	1.25
EXP19	1534	512	0.0005	0.0014	2.80
EXP21	3070	1024	0.0013	0.0024	1.85
EXP23	6142	2048	0.0016	0.0210	13.12
EXP25	12 286	4096	0.0030	0.0505	16.83
EXP27	24 574	8192	0.0094	0.1301	13.84
EXP29	49 150	16 384	0.0120	0.4581	38.18
EXP31	98 302	32 768	0.0203	2.9610	145.86
EXP33	196 606	65 536	0.0369	14.2462	386.08
EXP35	393 214	131 072	0.0658	69.3435	1053.85
EXP37	786 430	262 144	0.1161	344.8065	2969.91
EXP39	1 572 862	524 288	0.2161	1556.8613	7204.36
EXP41	3 145 726	1 048 576	0.4172	-	-
EXP43	6 291 454	2 097 152	0.8165	-	-
EXP45	12 582 910	4 194 304	1.8365	-	-
EXP47	25 165 822	8 388 608	3.1044	-	-
EXP49	50 331 646	16 777 216	5.7862	-	-
EXP51	100 663 294	33 554 432	11.2031	-	-
EXP53	201 326 590	67 108 864	22.2010	-	-
EXP55	402 653 182	134 217 728	43.5169	-	-
EXP57	805 306 366	268 435 456	87.3454	-	-
EXP59	1 610 612 734	536 870 912	174.8520	-	-
EXP61	3 221 225 470	1 073 741 824	385.8030	-	-

Table 3: Martins vs. MDA on the EXP instance set.

is a scatter plot representation of the running times in the table depending on the graphs' number of nodes. The MDA clearly outperforms Martins's algorithm on this set of instances. The biggest such instance that Martins's algorithm manages to solve is the one with 39 nodes. It does so in 1556.86s while the MDA solves this instance in 0.22s. This results in a speedup of approximately ×7204. The speedup is proportional to the graph size increases. The effect can be clearly seen in Figure 4. In the table and in the plot we can also observe that the MDA manages to solve EXP instances with up to 61 nodes. This instance is solved in 385.8s, indicating that the algorithm would possibly solve bigger instances if we would assign more memory. However, the huge sets of minimal complete sets of efficient paths from the EXP63 onward causes the MDA runs to fail in our experimental setting.

All in all we conclude that on these artificial instances, our implementations mirror the asymptotic behavior of both algorithms. The solution time correlates with the number N of permanent paths. Then, in Figure 4 the logarithmic y-axis causes the scatter plots to look linear w.r.t. n and w.r.t. m (since for every value of n there is only one value of m on this set of instances). Since the propagation of new paths is a $\mathcal{O}\left(dmnN_{\max}^2\right)$ operation in

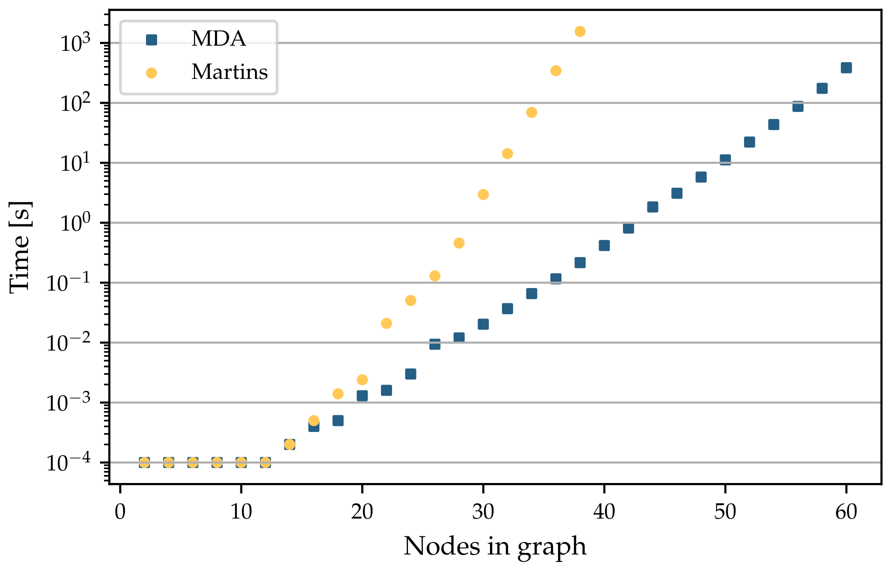


Figure 4: Running times of Martins's algorithm and the MDA on 3d EXP instances.

Martins's algorithm and a $\mathcal{O}\left(dmN_{max}^2\right)$ operation in the MDA (cf. Table 1), it is to be expected that the data series representing the running times of Martins's algorithm has a steeper slope in Figure 4. The high computational cost caused by the merge operation required in every iteration of Martins's algorithm unveils, particularly on the set of EXP instances, the drawback of this algorithm.

In the following sections we will see that while the MDA is faster than Martins's algorithm, the speedup is considerably smaller than the speedup reached on EXP instances. This observation is crucial at this point as it dispels any concerns about whether the results presented in Table 3 are solely a consequence of inefficient implementation in Martins's algorithm.

6.3.2 Grid Instances

In Table 4 we present the summarized results for grid graphs. Every row in the table reports the results of both algorithms depending on the number N of permanent paths. For every instance, the number N of permanent paths at the end of the algorithms coincides in the MDA and in Martins's algorithm. The i^{th} row, $i \in \{1, \ldots, 7\}$, of the table collects the instances for which the algorithms made at least $10^i + 1$ and at most 10^{i+1} paths permanent.

The first row is irrelevant: both algorithms solve the instances with at most 100 permanent paths in less than 0.1 milliseconds. Instances with at most 10^4 permanent paths are solved faster by Martins's algorithm. The reason is that the number of explored paths stored simultaneously during the solution process is not high and thus, the merge operations require fewer comparisons than the repeated dominance checks in the nextQueuePath routine. See also Section 6.1.

The situation changes in favor of the MDA for instances with more than 10^4 permanent paths. The speedup grows steadily and reaches $\times 3.27$ on the 310 instances that require between 10^7 and 10^8 permanent paths to be

N interval	Instances	N	N_t	time mda	time martins	Speedup
$(10^1, 10^2]$	2	32.56	1.41	<0.0001	<0.0001	1
$(10^2, 10^3]$	6	439.69	5.16	0.0002	0.0002	0.92
$(10^3, 10^4]$	32	3240.75	20.38	0.0018	0.0016	0.88
$(10^4, 10^5]$	72	37 602.31	92.44	0.0246	0.0276	1.12
$(10^5, 10^6]$	188	369 126.73	355.01	0.4192	0.5827	1.39
$(10^6, 10^7]$	390	3 594 007.97	1782.16	9.9572	26.3608	2.65
$(10^7, 10^8]$	310	20 341 258.55	8085.47	161.4992	528.4312	3.27

Table 4: Martins vs. MDA on Grid instances. Both algorithms solve all instances.

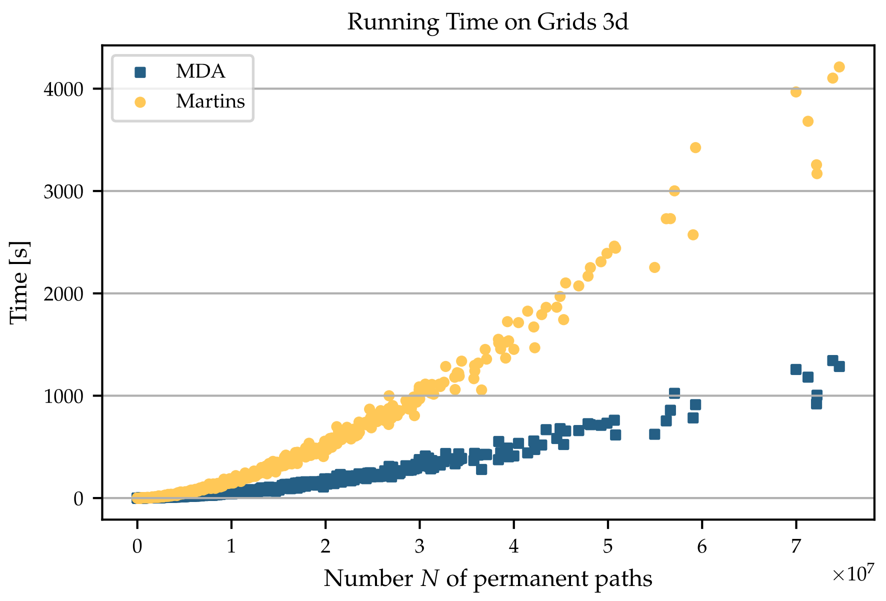


Figure 5: Running times of Martins's algorithm and the MDA on 3d Grid instances using linear axes.

solved. On these instances, a minimal complete set of efficient s-t-paths contains 8085.47 paths on average.

Figure 5 and Figure 6 show scatter plots comparing the running times of both algorithms depending on the number N of permanent paths. The log-log-plot in Figure 6 is particularly interesting since it shows that the behavior of both algorithms' implementations mirrors the asymptotic bounds from Table 1: for $d = 3$ the difference between both algorithms is a multiplicative factor n and almost vanishes in the log-log-plot.

Finally, in Figure 7 we plot the extractions per second of both algorithms again depending on N. It is clear that this measure for the algorithms' efficiency decays as N grows because every $\preceq_D$-check and every merge operation becomes harder to solve with increasing output size. The plot shows that the efficiency of the MDA decays slower and that after a threshold (approximately 4×10^7) the difference between both algorithms remains stable.

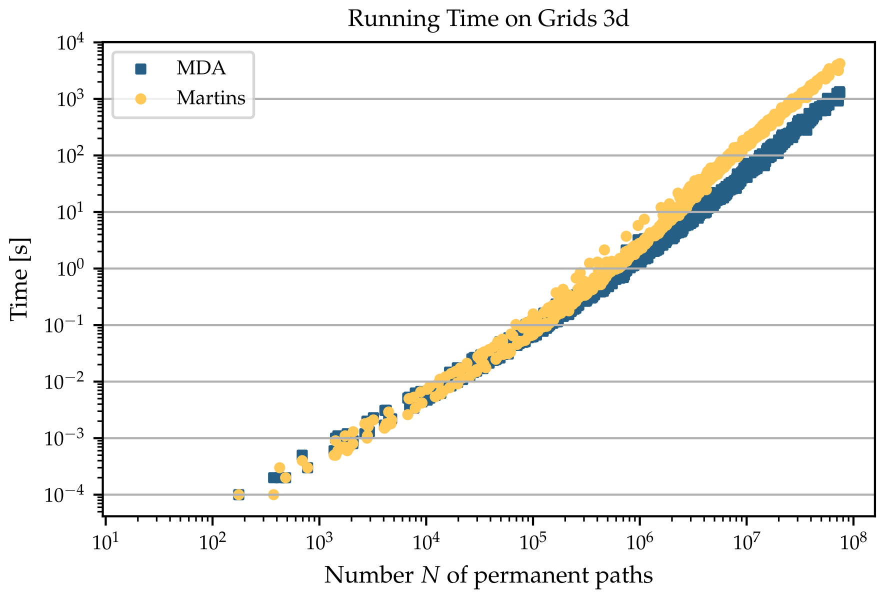


Figure 6: Running times of Martins's algorithm and the MDA on 3d Grid instances using logarithmic axes.

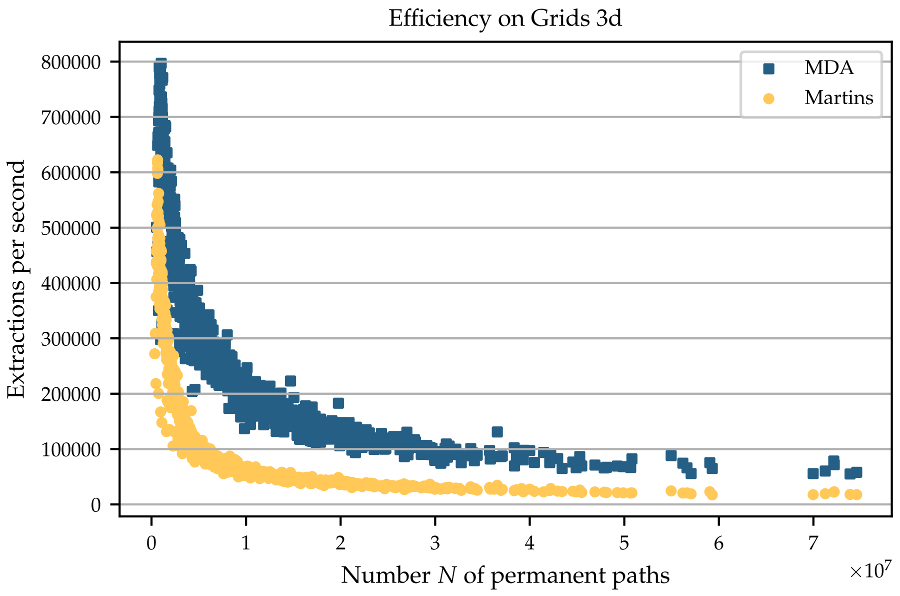


Figure 7: Comparison of the efficiency of Martins's algorithm and the MDA on 3d Grid instances.

6.3.3 NetMaker Instances

In Table 5 we summarize the results obtained on NetMaker instances. The table's structure is similar to Table 4. In this case, every row contains the results obtained by both algorithms on graphs with the same number of nodes. Both algorithms solved every instance within the time limit.

NetMaker instances are notably different to Grid instances regarding the output sizes N and N_t w.r.t. the graphs' size. The last line in Table 4 encodes the biggest Grid instances regarding both the number of nodes in the graph and the instances' output size. Dividing N by the number of nodes in the grid, i.e., $n = 10000$, we obtain an average number of 2034.13 permanent paths per node. On NetMaker instances we can see in the last line of Table 5 that instances with $n = 30000$ require the greatest value N of permanent paths to be solved and, on average, only produce 507 permanent paths per node. The smaller input graphs with 5000 nodes produce around $1200/5 = 240$ permanent paths per node. Thus, on average, six times smaller graphs regarding the number of nodes produce half as many permanent paths per node. This explains why the speedup achieved by the MDA is slightly bigger on smaller graphs $\times 1.82$ for graphs with 5000 nodes than for the biggest graphs where it reaches $\times 1.5$. All in all, the speedups in favor of the MDA are lower than on grid graphs because the size of the P^*_{sv} lists is much smaller. NetMaker instances are easier to solve than Grid instances.

In Figure 9 we plot the running times of both algorithms w.r.t. the number of permanent paths on NetMaker graphs with 30000 nodes. The same plot looks very similar for the instances encoded in the other rows of Table 5. For this reason, we do not include them in the printed version of the thesis, but they are all contained in the directory results/plots in the repository (Maristany de las Casas, 2023a). The naming of the files in the directory is self-explanatory.

Furthermore, NetMaker instances are interesting because the varying sizes of the graphs result in different graph densities (number of arcs divided by number of nodes). Table 6 and Figure 8 summarize the results obtained from NetMaker instances separating and plotting them depending on the graphs' density. To group the graphs, we rounded their density to the next integer. We observe that increasing density of the graph impacts the algorithms' efficiency similarly. Moreover, the red data shows that the speedup reaches values between $\times 1.9$ and $\times 2$ as the densities grow up to 17. For densities equal to 21 and 23, the speedup decays to a value of around $\times 1.7$. This is because the numbers N and N_t for these groups of instances indicate that the cardinality of the lists of permanent paths is smaller on these graphs than on graphs with density 17. Thus, merge operations and $\preceq_D$ -checks can be performed faster. Indeed, the speedups in favor of the MDA are proportional to the average number of permanent paths N.

6.3.4 Road Instances

The results obtained on road instances are summarized in Table 7. These instances are notably hard and by now, we cannot claim that they are a solved instance class for 3-dimensional arc costs. Every row in the table reports the

				time		
netM-n	Instances	N	N_t	mda	martins	Speedup
$n = 5$	240	1 197 681.03	695.66	6.6550	12.1276	1.82
$n = 10$	220	3 398 739.57	941.42	30.7094	48.0459	1.56
$n = 15$	240	6 639 691.63	1126.15	78.3546	122.1306	1.56
$n = 20$	240	9 868 444.80	1252.33	132.7924	204.8363	1.54
$n = 25$	200	11 313 273.81	1125.89	153.4810	227.2383	1.48
$n = 30$	240	15 213 764.84	1344.04	230.8244	346.3387	1.50

Table 5: Martins vs. MDA on the NetMaker instances. Both algorithms solved all instances within the time limit. Instances are grouped depending on the number of nodes in the input graph.

				time		
Density	Instances	N	N_t	mda	martins	Speedup
6	240	1 927 527.91	319.27	7.1516	8.7192	1.22
8	240	3 388 611.54	605.15	20.3446	28.1831	1.39
11	220	5 409 865.60	994.40	52.4636	81.0655	1.55
13	240	7 310 322.13	1311.05	95.0760	154.1649	1.62
16	100	14 126 390.82	2905.09	299.7726	591.6235	1.97
17	120	19 265 540.88	3362.51	531.3614	1026.5097	1.93
21	100	11 341 181.08	1975.62	258.5909	439.6957	1.70
23	120	11 478 804.80	2070.82	289.8834	505.3561	1.74

Table 6: Martins vs. MDA on the NetMaker instances. Both algorithms solved all instances within the time limit. Instances are grouped depending on the density of the input graph.

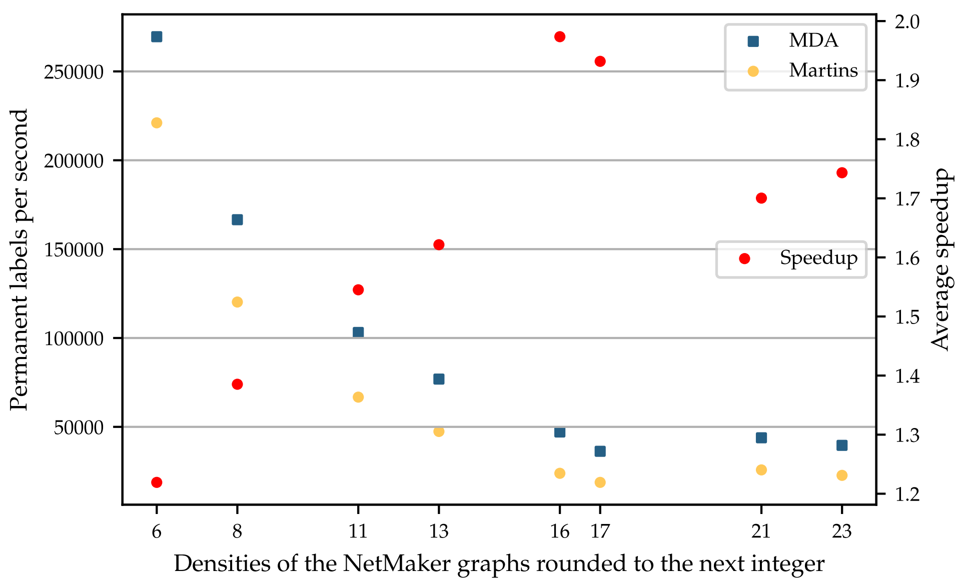


Figure 8: Comparison of the efficiency of Martins's algorithm and the MDA on NetMaker instances. As the efficiency metric we use the paths that are made permanent per second. The instances are grouped depending on the graphs' density. The y-axis on the right hand side and the red data series show the average speedup on these groups of instances.

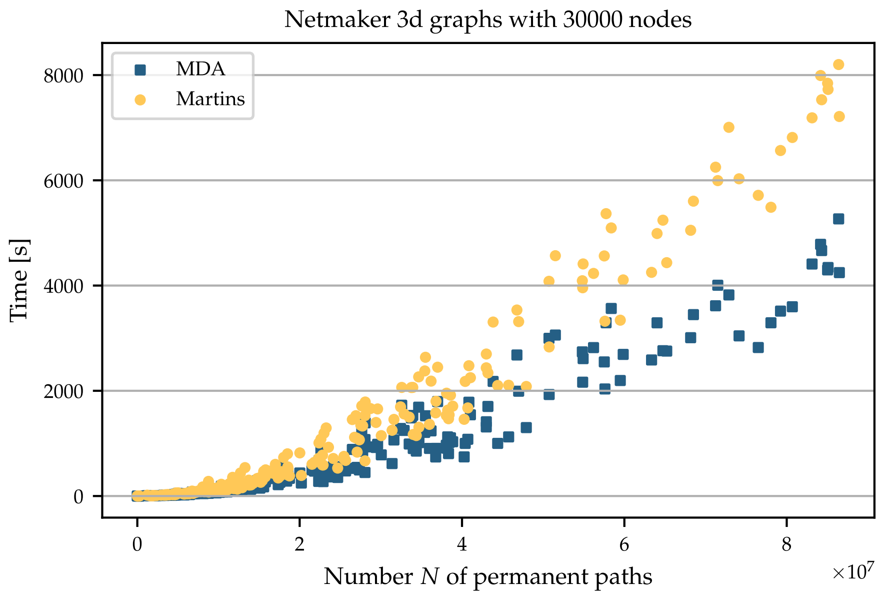


Figure 9: Running times of Martins's algorithm and the MDA on 3d NetMaker instances with 30000 nodes.

collected results for the instances defined on a single road network. Thus, 100 instances are encoded per row but even for the smallest NY network, none of the algorithms solves all instances within the time limit. While the MDA manages to solve 73 instances, Martins's algorithm solves only 56 on this network. Hence, for road networks we report averages as explained in Remark 6.2.

Since the MDA always solves at least as many instances as Martins's algorithm, the reported numbers in the N and N_t columns are never smaller for the MDA. Sticking to the NY network, we observe that the MDA managed to solve instances with more than 19 million permanent paths while the instances solved by Martins's algorithm had slightly more than 8 million permanent paths on average.

The average running times reported in Table 7 are surprising because they are not close to the time limit. The reason can be seen in Figure 10-Figure 15 and in the results files in (Maristany de las Casas, 2023a) which contain the data collected for every instance separately. In these scatter plots the running time needed to solve the instances considered in Table 7 are plotted w.r.t. the number of permanent paths. The yellow dots on the $y = 7200$s line are easy to distinguish and they correspond to instances solved by the MDA and not by Martins's algorithm. We can see that many of the 100 instances defined for every road network are solved in less than 1000s. While the sets of permanent s-t-paths are rather small compared to the sets of s-t-paths on Grid instances and on NetMaker instances, the size of the road networks makes the handling of the solution sets challenging. This causes a very fast increase of the number of permanent paths required to solve the instances. Thus, only few instances lie in the imaginary interval between *easy to solve* ($<$ 1000s) and *not solvable in* 7200s. In fact, in Figure 12-Figure 15 a clustering of the instances' running times becomes increasingly clear. In Figure 15 we observe a cluster of instances around $N = 1.0 \times 10^8$ and then

	MDA				Martins				
	solved	N	N_t	time	solved	N	N_t	time	Speedup
NY	73	19 070 244.84	466.21	71.51	56	8 032 884.33	255.22	126.22	1.77
BAY	65	20 191 649.61	579.89	83.53	60	16 091 057.25	484.84	112.60	1.35
COL	46	19 235 827.47	437.75	104.31	40	12 268 279.58	331.59	181.52	1.74
FLA	17	10 898 652.80	212.08	38.33	15	7 390 904.21	166.57	70.31	1.83
NE	13	29 552 102.94	843.67	136.61	13	29 552 102.94	843.67	202.91	1.49
LKS	8	27 515 810.26	718.88	146.85	8	27 515 810.26	718.88	249.01	1.70

Table 7: Martins vs. MDA on road instances. For every road network, we defined 100 s-t-pairs. Instances that are not solved by any of the algorithms are ignored. There are no instances solved only by Martins's algorithm. For the time and speedup columns we assume Martins's algorithm to solve the instances solved only by the MDA in 7200s. The averages for N and N_t only consider solved instances.

no more instances with $N < 2.0 \times 10^8$. As a result, the reported solution time averages are not close to 7200s as one would expect. The choice of reporting the geometric mean in the time-columns also lowers the reported numbers.

For now, we can point out that despite having solvability issues, the MDA is consistently faster than Martins's algorithm also on Road instances. On the NY network, the one on which the MDA solves the most instances, the speedup is $\times 1.77$. The smallest speedup, $\times 1.33$, is achieved on the BAY network, where the MDA solves 65 instances and Martins's algorithm solves 60 instances, its maximum. The biggest speedup, $\times 1.83$ is achieved on the FLA network. On the NE and the LKS network, the achieved speedup is in line with the one achieved on the other road networks. Solving 13 and 8 out of 100 instances on these graphs is arguably not a lot. Thus, we refer the reader to Chapter 9, where versions of the MDA and of Martins's algorithm tailored for large scale One-to-One MOSP instances perform better.

Note that in the original publication of the MDA (Maristany de las Casas, Sedeño-Noda, & Borndörfer, 2021), we achieved average speedups of more than $\times 5$ when comparing the MDA with the improved version of Martins's algorithm from (Demeyer et al., 2013). However, the algorithm as introduced in Chapter 5 is faster, causing smaller speedups in this chapter of the thesis.

Remark 6.3. *First Guerriero and Musmanno (2001) and later Paixão and Santos (2013) claimed that label-correcting strategies are better suited than label-setting strategies in the design of MOSP algorithms. The label-setting setup in both papers uses the algorithm by (E. Q. V. Martins, 1984) storing all explored paths in a heap, a list, or a deque data structure simultaneously. The sorting keys are either the lexicographic order or the sum-of-costs order. For the benchmarks using the lexicographic order, the dimensionality reduction technique (cf. Section 3.5) is not used. They use MOSP instances with up to ten-dimensional arc costs in their experiments. Such an extensive analysis has not been conducted using modern label-setting MOSP algorithms. In this thesis we are interested in the suitability of the MOSP model to capture large scale real world applications. If we use an implementation of Martins's algorithm that stores all explored paths simultaneously in a priority queue, that performs merge operations on the elements in that queue, and that does not use dimensionality reduction, the resulting running times are orders of magnitude longer than the ones reported in this chapter. In that scenario label-correcting methods can win the comparison. However, the speedup obtained*

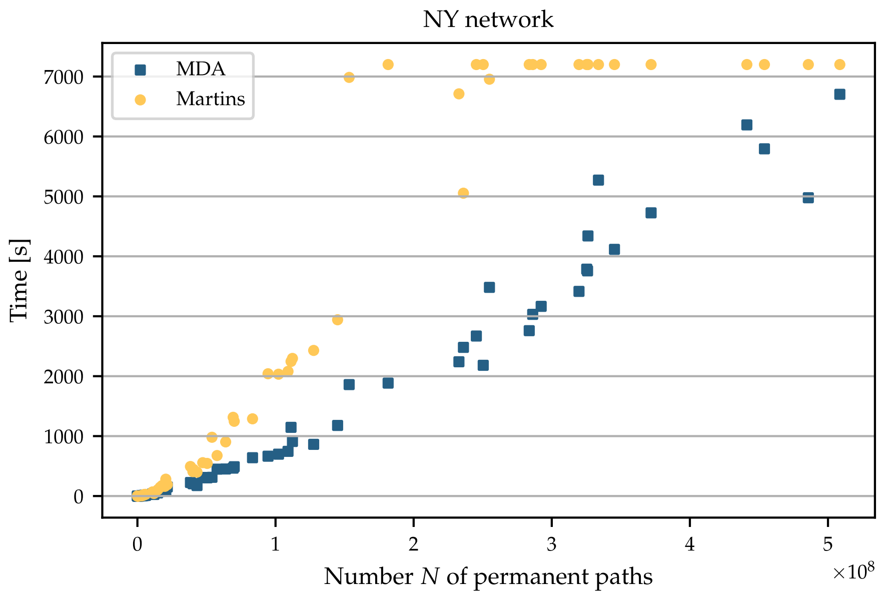


Figure 10: Running times of Martins's algorithm and the MDA on 73 NY instances.

using such algorithms in (Guerriero & Musmanno, 2001) and in (Paixão & Santos, 2013) is not comparable with the speedup obtained in our early tests that led to the design of Martins's algorithm as introduced here.

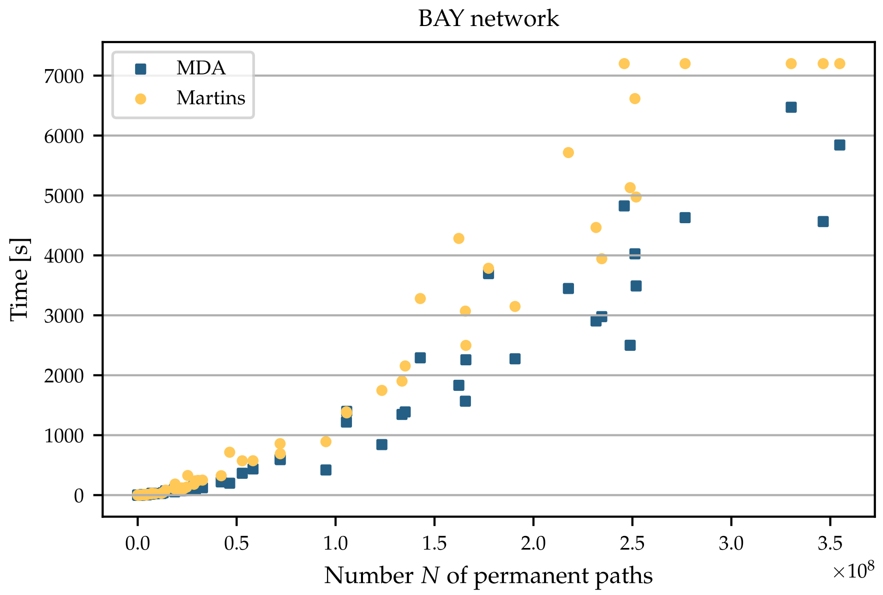


Figure 11: Running times of Martins's algorithm and the MDA on 65 BAY instances.

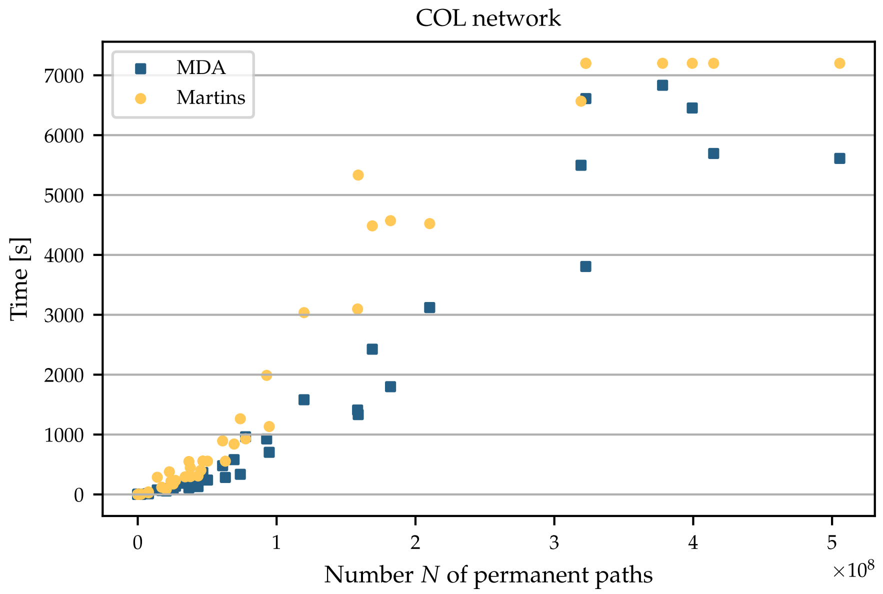


Figure 12: Running times of Martins's algorithm and the MDA on 46 COL instances.

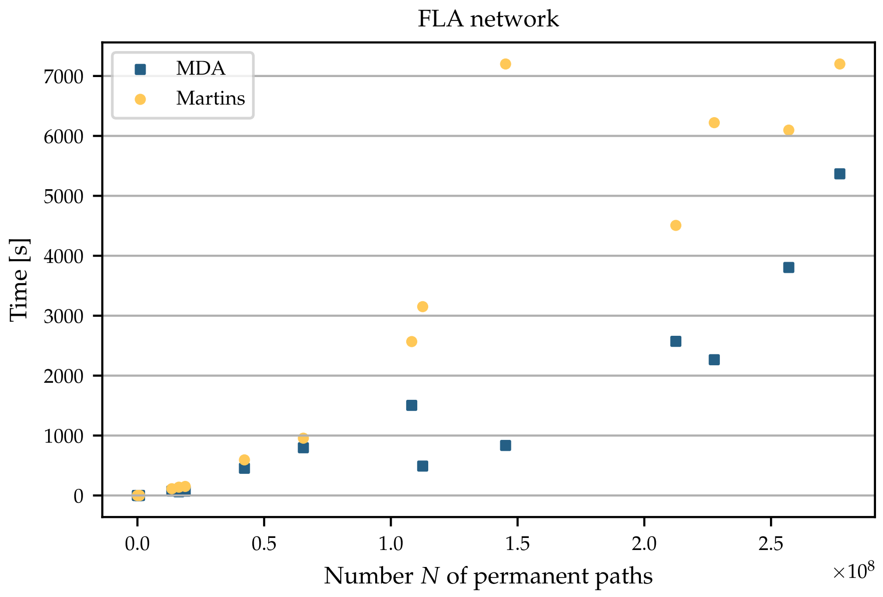


Figure 13: Running times of Martins's algorithm and the MDA on 17 FLA instances.

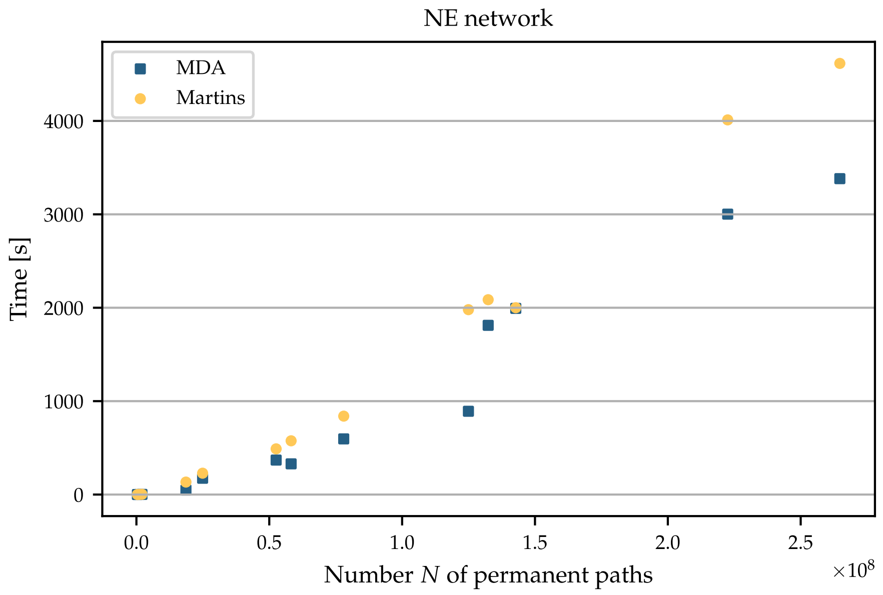


Figure 14: Running times of Martins's algorithm and the MDA on 13 NE instances.

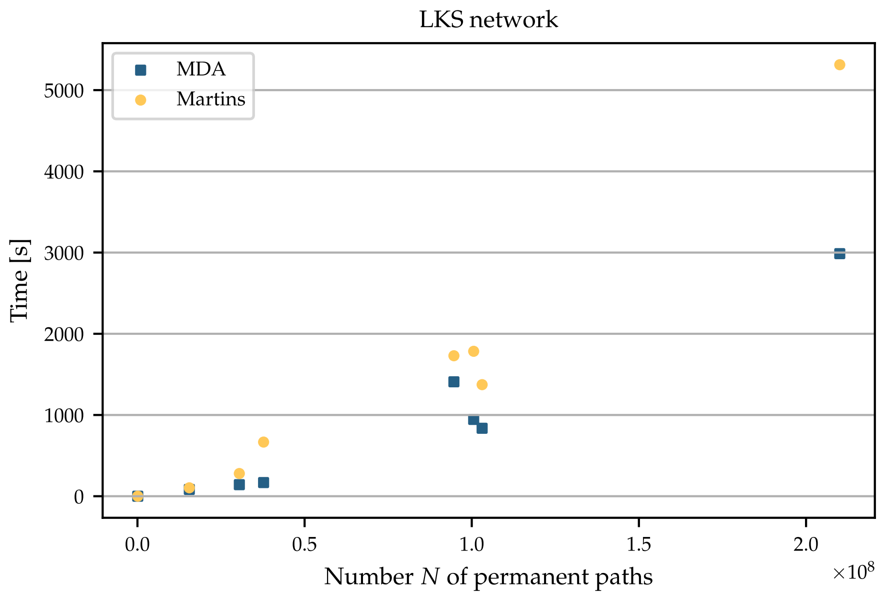


Figure 15: Running times of Martins's algorithm and the MDA on 8 LKS instances.

7 CONCLUSION

The experiments in the last chapter end this part of the thesis. The main contribution is the introduction of the *Multiobjective Dijkstra Algorithm* (MDA), a label-setting algorithm for the Multiobjective Shortest Path (MOSP) problem. The key to design good MOSP algorithms is the efficient handling of *explored paths*. For a label-setting MOSP algorithm a path is an explored path if it has already been constructed but the algorithm is not able to provably discard the path or store it as part of the output because it is efficient. The main novelty in the MDA is that it exploits the Dynamic Programming nature of the MOSP problem to store at most one explored path per node in the input graph. The price for doing so is that paths need to be reconstructed from efficient subpaths and that $\preceq_D$ -checks (checks to decide whether an explored path is dominated by or equivalent to an efficient path with coincident end nodes) for a given explored path might need to be repeated. The crux is that rebuilding paths is asymptotically negligible and that we can bound the amount of times $\preceq_D$-checks are repeated. By doing so, we can prove that the overall asymptotic running time and space consumption bounds of the MDA improve the known output sensitive bounds for other label-setting MOSP algorithms.

Our journey to end up with the version of the MDA described in this chapter guided us also through incremental speedup techniques from the literature. The *dimensionality reduction* (cf. Section 3.5) in conjunction with the lexicographic ordering of the explored paths to speedup the costly $\preceq_D$-checks, the usage of a *memory pool* in our implementations to enhance memory management, and the *target pruning* to discard more paths in One-to-One scenarios are prominent examples. Including them in the MDA ensures that besides the theoretical improvement, the new algorithm is fast in practice.

To benchmark this aspect, we added Martins's algorithm to the mix. The algorithm was first published in (E. Q. V. Martins, 1984) and is undeniably the classical label-setting MOSP algorithm. We included all speedup techniques mentioned in the previous paragraph in our version of the algorithm. Moreover, we slightly tuned the handling of explored paths to avoid inefficiencies that would make the head to head comparison with the MDA unfair. This results in a new version of the original Martins's algorithm with better asymptotic complexity and also better performance in practice. While Martins's algorithm does not rebuild paths and it does not repeat $\preceq_D$-checks for a given explored path, it needs to perform costly merge operations in every iteration. These subroutines act on the set of explored paths for a fixed node and ensure that no two explored paths dominate each other or a cost-equivalent. This requires a linear time effort w.r.t. to the number of explored paths. If N_{max} is the number of the largest set of efficient paths stored by a label-setting MOSP algorithm for a given instance, there can be nN_{max} explored paths for a fixed node. On the other hand $\preceq_D$-checks require a linear time effort w.r.t. N_{max} only. Thus, the $\mathcal{O}(nN_{max})$ merge operations stand in contrast to the repeated $\preceq_D$ checks in $\mathcal{O}(N_{max})$ in the MDA. The main takeaway of this part of the thesis is that asymptotically and in practice, the avoidance of merge operations in the MDA is the way to go in the design of label-setting MOSP algorithms that also adhere to a possibly tight memory consumption bound.

Part III

THE MULTIOBJECTIVE DIJKSTRA ALGORITHM IN PRACTICE

8 INTRODUCTION

After publishing (Maristany de las Casas, Sedeño-Noda, & Borndörfer, 2021) and introducing the MDA, one thing was clear, particularly looking at the experimental results on road networks: the MDA is not the answer if the goal is to establish the flexibility of modeling real world shortest path problems with higher-dimensional arc costs as a go-to choice for practitioners.

In this part of the thesis, driven by our cooperation with Lufthansa Systems GmbH, we design variants of the MDA that aim to bring the techniques described already in the thesis closer to usability in practice. In Chapter 9 we design the *Targeted Multiobjective Dijkstra Algorithm*, an A*-like One-to-One MOSP algorithm. It stores more explored paths than the MDA simultaneously but it has the same asymptotic running time behavior. The content of this chapter was published in (Maristany de las Casas, Kraus, et al., 2023).

Chapter 10 and Chapter 11 are the embedding of (Maristany de las Casas, Borndörfer, et al., 2021) into this thesis. In Chapter 10 we discuss how to add *time dependent* arc cost functions to MOSP models. The generalization of the Time Dependent Shortest Path problem to the multiobjective scenario follows an intuitive path but we encounter remarkable limitations and strict conditions on the arc cost functions that restrict the types of practical problems that can be modeled using the *Dynamic Multiobjective Shortest Path* (Dyn-MOSP) problem. An in depth analysis of the problem was missing in the literature. Given the relevance of arc cost functions that are dependent on e.g., the time point at or the weight with which the arc is traversed, our discussion in Chapter 10 is the second contribution in this part of the thesis.

However, modeling using multiple objectives can lead to output sets that are too large for decision makers to take advantage of the new solution format. This opens up the search for approximation algorithms for MOSP that return a meaningful representation of the output in the exact case. In Chapter 11 we discuss a new *FPTAS* for MOSP based on the MDA. At the end of this chapter, we also discuss our experimental results on instances of the *Horizontal Flight Planning* problem made available by our industry partner Lufthansa Systems GmbH. These instances are Dyn-MOSP instances that, when solved with an exact Dyn-MOSP algorithm cause too many paths to be efficient.

Remark 8.1. *The storytelling in this introduction does not mirror the chronology of the publications that led to this part of the thesis. This is why the experimental results at the end of Chapter 9 and at the end of Chapter 11 are not directly comparable. The results in Chapter 9 are obtained using our newest implementation of the T-MDA that is publicly available and not tailored to the flight planning application. In contrast, the results presented in Chapter 11 are obtained from an older implementation of the MDA that relies on libraries and some techniques that prepare the code to be easily transmitted to the optimization core of Lufthansa Systems. However, the* *effects that both sets of experiments want to show are different and not necessarily incremental which is why we keep the described exposition in the thesis.*

9 ONE-TO-ONE MULTIOBJECTIVE SHORTEST PATH PROBLEM

To study the One-to-One MOSP problem (cf. Definition 3.2) in more detail, we embed the content first published in Maristany de las Casas, Kraus, et al. (2023) in this chapter. Let $\mathcal{I} = (G = (V, A), s, t, d, c)$ be a One-to-One MOSP instance. Out of the box, the MDA and Martins's algorithm solve $\mathcal{I}$. On the asymptotic side of the algorithm design, we do not achieve an improvement in this chapter. Even worse, as already mentioned in Section 3.2, in (Bökler, 2018, Theorem 6.2) the author proves that the One-to-One MOSP problem is not output sensitive unless $\mathcal{P} = \mathcal{NP}$. Thus, our effort in this chapter focuses on the design of a variant of the MDA for One-to-One MOSP instances that matches the MDA's running time bound derived in Theorem 4.4 but is tuned for performance in practice. Thereby, we discuss three improvements.

1. As it is usual in the single-criterion A^* algorithm, we can guide the search towards the target using node potentials. We discuss them in Section 9.3.

2. For the experiments with One-to-One MOSP instances in Chapter 6 we already explained *target pruning*, additional $\preceq_D$-checks used to discard s-v-paths for $v \in V \setminus \{t\}$ if they are dominated by or equivalent to permanent s-t-paths. The effect of this technique is enhanced in the presence of node potentials. In Section 9.4, we elaborate on this technique and show how we combine it with the MDA.

3. The last point modifies the MDA and leads to the *Targeted Multiobjective Dijkstra Algorithm (T-MDA)* (Section 9.5). It uses a novel *pseudo-lazy management* of explored paths that increases the memory consumption of the T-MDA compared to the one of the MDA but allows the T-MDA to match the asymptotic running time bound of the MDA.

The contributions are explained in more detail in Section 9.2. Before that, we give an overview of the One-to-One MOSP literature landscape.

9.1 LITERATURE REVIEW

The A^* label-setting algorithm for One-to-One Shortest Path problems (Hart et al., 1968; Goldberg et al., 2006) gets a *node potential* as part of its input. These potentials, often called *heuristics* in the literature, underestimate the optimal cost from a node to the target node. Then, every path is evaluated based on the sum of its exact cost and the heuristic value associated with the path's end node. Hence, heuristics alter the priority with which paths are processed, benefiting paths with good heuristic values, i.e., paths that are expected to reach the target along a cheap extension. First Stewart and White (1991) and later Mandow and Pérez de la Cruz (2005, 2010) generalized this notion to the multiobjective scenario. Even later and with Pulido, F. J. as a new co-author they introduced the NAMOA* algorithm (Pulido et al., 2014). An improved version called NAMOA$^*_{\mathrm{dr}}$ algorithm (Pulido et al., 2015), is still the state of the art One-to-One MOSP algorithm in the literature. Its main performance improvement w.r.t. the original NAMOA* algorithm is the use of the dimensionality reduction technique that we discussed in Section 3.5. The impact of the NAMOA$^*_{\mathrm{dr}}$ for today's MOSP algorithms is remarkable. Nowadays all fast MOSP algorithms include the dimensionality reduction technique. Moreover, already in (Mandow & Pérez de la Cruz, 2010) the authors proved that their NAMOA* algorithm is optimal regarding the number of expanded explored paths.

The MDA and its asymptotic running time bound derived in Theorem 4.4 prove that the management of explored paths (cf. Definition 3.8) is of utmost relevance to design efficient MOSP algorithms. Consider a One-to-One MOSP instance (G, s, t, d, c) as in Definition 3.2 and assume that all algorithms use a priority queue called Q to store explored paths. Then, the three main ways to handle explored paths in the literature are the following.

1. For any node $v \in V$, the MDA stores at most one explored s-v-path p at a time in Q. No other explored paths are stored. Explored paths that were seen but did not match the criteria to enter the queue are rebuilt as needed exploiting the subpath efficiency of efficient paths (cf. Theorem 3.1). In this approach the number of explored paths that are stored simultaneously is minimal.

2. The classical label-setting algorithm by Martins (E. Q. V. Martins, 1984) stores more explored paths as discussed in Chapter 5. For every node $v \in V$ a list of explored s-v-paths exists. In any of these lists, no element dominates another. Maintaining this property is achieved using merge operations. Efficient implementations of Martins's algorithm like our Martins's algorithm store only a minimal (w.r.t. $\prec_Q$) path from every such list in the algorithm's priority queue.

3. Recently, Ulloa et al. (2020) and Ahmadi et al. (2021) proposed *lazy queue management* for One-to-One BOSP algorithms. The name *lazy* in this context is often used in the literature and is motivated by the fact that new explored paths are added to the algorithms' priority queue without performing a merge operation. In other words, algorithms using a lazy queue management insert new explored paths into Q without comparing them with any existing explored path. The price for lazy queue management is twofold. First $\preceq_D$-checks for every path extracted from the queue need to be repeated after their extraction. Second, the size of the priority queue can become an issue. The practical advantage is that the handling of every explored path is easy and does not require any additional data structures besides the algorithms' priority queue.

On a standard testbed of instances containing road networks of the United States with bidimensional integer arc cost vectors, Ahmadi et al. (2021) show that the theoretical drawbacks of lazy queue management are not a concern in practice. Ahmadi et al. (2021) overcome the problems using so called *bucket queues* (Denardo & Fox, 1979). These queues work well in the biobjective scenario with integer arc costs. The buckets are indexed according to the paths' first cost component. The second cost component remains unsorted within each bucket possibly causing extractions of paths in the wrong order. In theory, this violates the ordered extraction property (Lemma 4.4 for MDA and Lemma 5.2 for Martins's algorithm) that is needed to prove the correctness of label-setting MOSP algorithms. However, in the biobjective case the erroneous extractions can be easily detected and the erroneous propagation of the paths does not harm the practical performance of the algorithm. The size of the bucket heaps and the erroneous extractions, storage, and expansions make the mentioned algorithms asymptotically difficult to bound and worse than the MDA in this regard. As noted in the introduction of the thesis, we failed to design a biobjective version of the T-MDA that could beat the algorithm by Ahmadi et al. (2021) in the case of biobjective integer arc costs. In their work, Ahmadi et al. (2021) also discuss a bidirectional version of their algorithm.

In a general multiobjective scenario, less work has been published for One-to-One MOSP in recent years. This motivates our choice to design a variant of the MDA that uses a novel *pseudo-lazy management* of explored paths. The running time of the new variant matches the running time of the original MDA and works for integer and rational arc costs using binary heaps.

9.2 OUR CONTRIBUTION

Having introduced the meaning of lazy queue management already, we can give a more detailed explanation of how we ended up designing the T-MDA. For every node $v \in V$ just one path is stored in the priority queue Q as in the MDA. However, the T-MDA also resembles lazy algorithms: every explored s-v-path p that is not inserted into the queue directly after becoming an explored path is not discarded and rebuilt later as in nextQueuePath. Instead, it is stored in a list of explored s-v-paths associated with the last arc, say $(u, v) \in A$, of p. This list of explored paths in the T-MDA contains all relevant explored s-v-paths whose last arc is (u, v).

The crux in the T-MDA is that using only constant time insertions (prepend and append) of paths to the new lists of explored paths the lists remain sorted w.r.t. the chosen total order $\prec_Q$. As a consequence, the size of the priority queue remains polynomially bounded in contrast to standard lazy queue management approaches (Ulloa et al., 2020; Ahmadi et al., 2021). Then, when an s-v-path is extracted from Q, the T-MDA finds the next queue path for v accessing the sorted lists of explored paths associated with arcs in $\delta^-(v)$. The paths in the lists being sorted allows us to solve (8) using the same asymptotic complexity as nextQueuePath but faster in practice because less dominance checks are performed overall and there is no need to rebuild paths (cf. Line 5 of nextQueuePath).

As mentioned earlier, the state of the art algorithm for One-to-One MOSP problems in general dimensions is the NAMOA* algorithm introduced in Pulido et al., 2015. It handles explored paths using merge operations as in

Martins's algorithm. Our second contribution in this chapter is the design of the *NAMOA$^*_{dr}$-lazy algorithm*, a variant of the NAMOA$^*_{\mathrm{dr}}$ algorithm that uses lazy queue management as described in Item 3 of the above enumeration. To the best of our knowledge such an algorithm is novel in the literature and generalizes the ideas from (Ulloa et al., 2020; Ahmadi et al., 2021) to the multiobjective case. Without this lazy queue management, the already improved NAMOA* from (Pulido et al., 2015) turns out not to be competitive against the T-MDA.

9.3 MULTIOBJECTIVE HEURISTICS

The definitions and results in this section generalize the A^* algorithm for single-criterion Shortest Path problems to the multiobjective case. The original A^* algorithm and its modern variants are discussed for example in the survey on the topic by Delling et al. (2009). The multiobjective generalization is also studied in the literature (e.g., Stewart & White, 1991; Mandow & Pérez de la Cruz, 2005).

Intuitively, we can think of *heuristics* as a node potential $\pi : V \to \mathbb{R}^d_{\geqslant 0}$. For a node $v \in V$, the node potential $\pi(v)$ lower bounds the cost of any v-t-path in the input graph. Then, for any s-v-path p considered during a label-setting MOSP algorithm, explored paths can be processed w.r.t. to the new costs $\bar{c}(p) := c(p) + \pi(v)$. The new paths' costs guide the search towards the target meaning that subpaths that are expected to reach the target node t with a better final cost are extracted from Q earlier.

The redefinition of the paths' costs mentioned in the last paragraph is only helpful if the efficient s-t-paths w.r.t. the original costs c coincide with the new costs $\bar{c}$. As in the single-criterion case, this is guaranteed if the heuristics fulfill the following conditions.

Definition 9.1 ((Admissible and Monotone) Heuristic. cf. Pulido et al., 2014)**.** A node potential $\pi : V \mapsto \mathbb{R}^d_{\geqslant}$ is *an admissible and monotone heuristic for $\mathcal{I}$*, if

admissibility:

i. $\pi(v) \leqslant c(p), \quad \forall v \in V,\ p\ v\text{-}t\text{-path}.$

monotonicity:

i. $\pi(t) = 0$, and

ii. $\pi(u) \leqslant c(a) + \pi(v), \quad \forall a = (u, v) \in A.$

Remark 9.1. *As in the single-criterion case, a monotone heuristic is admissible.*

Throughout the chapter, we only consider monotone and admissible heuristics. For the sake of brevity, we refer to them just as *heuristics* from now on. We always assume that a heuristic π is given.

Definition 9.2 (Reduced Costs of Arcs and Paths)**.** The *reduced costs* of an arc $a = (u, v) \in A$ are $\bar{c}(a) := c(a) + \pi(v) - \pi(u)$. For a u-v-path p between any two nodes u, $v \in V$, the *reduced costs of* p are

$$\bar{c}(p) := c(p) + \pi(v) - \pi(u) = \sum_{a \in p} \bar{c}(a). \tag{15}$$

Regarding the efficiency of paths, c and $\bar{c}$ are equivalent.

Lemma 9.1 (Efficiency equivalence)**.** *For any node $v \in V$ let $\bar{P}^*_{sv}$ be a minimum complete set of efficient s-v-paths w.r.t. $\bar{c}$. Then, $\bar{P}^*_{sv}$ is also a minimum complete set of efficient s-v-paths w.r.t c. The opposite also holds.*

Proof. The statement follows directly from (15) because the reduced cost of every s-v-path, alters the path's original cost w.r.t. the arc cost functions c by the same constant $\pi(v) - \pi(s)$. $\square$

We need to use monotone heuristics to be able to use label-setting MOSP algorithms to compute minimal complete sets of efficient s-t-paths using $\bar{c}$. This follows from the fact that for $a = (u, v) \in A$ the second monotonicity condition can be written as $0 \leqslant c(a) + \pi(v) - \pi(u)$ and the right hand side is precisely $\bar{c}(a)$. Thus, $\bar{c}(a) \geqslant 0$ for all $a \in A$ and no negative cycles in any cost dimension open up the possibility of efficient paths being non-simple or not well defined.

Theorem 9.1. *Consider a One-to-One MOSP instance $\mathcal{I} = (G, s, t, d, c)$ and a heuristic π. After defining $\bar{c}$ as in Definition 9.2, consider the One-to-One MOSP instance $\bar{\mathcal{I}} = (G, s, t, d, \bar{c})$.*

A label-setting MOSP algorithm solving $\mathcal{I}$ also solves $\bar{\mathcal{I}}$ and the returned minimal complete sets of efficient s-t-paths coincide up to cost-equivalent paths.

Proof. Follows directly from Lemma 9.1 and the nonnegativity of $\bar{c}$. $\square$

Note that label-setting algorithms only consider paths that start at the source node s. Thus, when prioritizing paths according to $\bar{c}$, we can neglect the $-\pi(s)$ term since all paths are equally influenced by it. Thus, from now on, we set the reduced costs of any s-v-path p, $v \in V$, as $\bar{c}(p) = c(p) + \pi(v)$.

For the used heuristics we have $\pi(t) = 0$. As a consequence, for s-t-paths p, we have $\bar{c}(p) = c(p)$. For any other path q ending at a node $v \in V \setminus \{t\}$, we have $c(p) + \pi(v)$ and $\pi(v) \geqslant 0$ by definition. Thus, s-t-paths are not penalized by the reduced costs while other paths possibly are. We know that paths in label-setting MOSP algorithms are extracted in non-decreasing order w.r.t. $\prec_Q$. Thus, s-t-paths are extracted in earlier iterations of the MDA while solving $\bar{\mathcal{I}}$ than while solving $\mathcal{I}$.

Computing Heuristics

Finding good heuristics is an application dependent task. This is particularly true for applications in which each arc cost component is not a scalar but a function that is evaluated depending on the path used to reach the arc's tail node. For example, the methodology to derive them in flight planning on airway networks (Blanco et al., 2022) and electric vehicle routing on road networks (Baum et al., 2020) varies and requires application specific techniques. There is however a generic way of computing heuristics and dominance bounds for d-dimensional One-to-One MOSP instances in polynomial time. This method is often used in the literature (e.g., Maristany de las Casas, Sedeño-Noda, & Borndörfer, 2021; Ehrgott, 2005; Ulloa et al., 2020; Ahmadi et al., 2021) and runs d queries of the Dijkstra's algorithm (Dijkstra, 1959). These queries are ran from the target node t to all other nodes (One-to-All Dijkstra) on the reversed digraph of G. Then, the i^{th} query, $i \in \{1, \dots, d\}$, returns a shortest path tree that contains, for every $v \in V$ for which there exists a v-t-path in G, a shortest v-t-path $p_{i,v}$ w.r.t. the arc costs $c_i(a)$, $a \in A$. We then set $\pi_i(v) = c(p_{i,v})$ for every $i \in \{1, \dots, d\}$. Since the original arc costs are positive, $\pi(t) = 0$, which is the first condition for the monotony of π. For any arc $a = (u, v) \in A$, the second condition required for the monotony of π is $\pi(u) \leqslant c(a) + \pi(v)$. Note that in any cost dimension $i \in \{1, \dots, d\}$, $c_i(a) + \pi_i(v)$ is the cost of a path from u to t using the arc (u, v) first and then a shortest v-t-path. On the other side of the inequality, the cost $\pi_i(u)$ is, by construction, the cost of a shortest u-t-path w.r.t. the arc costs c_i. Hence, $\pi_i(u) \leqslant c_i(a) + \pi_i(v)$ holds.

Recall that heuristics are admissible, i.e., $\pi(v) \leqslant c(p)$ for any v-t-path p. In this sense, the heuristic π computed during the d Dijkstra queries mentioned above is an optimal heuristic: increasing $\pi(v)$ in any dimension $i \in \{1, \dots, d\}$ causes a v-t-path p with minimal $c_i(p)$ cost component to violate the admissibility of π.

In the T-MDA we assume that a heuristic is part of the input. In our computational experiments, we compute π as described in the previous paragraphs during a preprocessing phase.

9.4 PRUNING

The true benefit of starting to make s-t-paths permanent earlier as discussed in the last section is that it enables us to *prune* irrelevant paths more effectively. A pruning rule in the context of label-setting MOSP algorithms is an additional $\preceq_D$-check that if answered positively, allows us to provably discard a (sub)path because its expansion towards the target cannot produce an efficient s-t-path.

Lemma 9.2 (Target Pruning)**.** *For a node $v \in V$ let p be an s-v-path built in Line 2 of propagate or in Line 5 of nextQueuePath. In case $c(P^*_{st}) \preceq_D \bar{c}(p)$, p can be discarded because the concatenation of p with any v-t-path q in G results in an s-t-path $p \circ q$ that is either dominated by or cost-equivalent to a path in P^*_{st}.*

Proof. We consider only monotone heuristics. They are also admissible. This gives us $\pi(v) \leqslant c(p)$ for any v-t-path in G. Given that $\bar{c}(p) = c(p) + \pi(v)$ the admissibility of π proves the statement. □

Up to now, we have discussed how to reformulate a given One-to-One MOSP instance to obtain a new one using a given heuristic. The two benefits of using the MDA to solve the new instance are on one hand the improved priority of s-t-paths and on the other hand the possibility to prune irrelevant paths to avoid their expansion. In the next section we change how the MDA actually works to design a new version of the algorithm that is more efficient in practice.

9.5 TARGETED MULTIOBJECTIVE DIJKSTRA ALGORITHM

In this section, we introduce and analyze the *Targeted Multiobjective Dijkstra Algorithm* (T-MDA). The pseudocode is shown in Algorithm 7. Making use of the fact that the MDA is known, we outlined the changes in the T-MDA already in Section 9.2. In this section, we discuss the details. We assume that the total order $\prec_Q$ used to sort explored paths is the lexicographic order. We use the order in conjunction with the dimensionality reduction technique, which we include explicitly in the pseudocode.

INPUT AND OUTPUT The input of the T-MDA is a d-dimensional One-to-One MOSP instance (G, s, t, d, c) and a heuristic π. In the description of the algorithm, we use the costs $\bar{c}$ as described in Section 9.3. The output is a minimal complete set of efficient s-t-paths P^*_{st}.

Algorithm 7: Targeted Multiobjective Dijkstra Algorithm

```
Input  : d-dimensional One-to-One MOSP instance $\mathcal{I} = (G, s, t, d, c)$,
         heuristic $\pi$ for $\mathcal{I}$.
Output: Minimal complete set $P^*_{st}$ of efficient s-t-paths.
1  Priority queue of paths $Q \leftarrow \emptyset$;                    // Sorted according to $\bar{c}$.
2  $\forall (u, v) \in A$ – explored s-v-paths that use $(u, v)$ as last arc: $\mathcal{NQP}_{uv} \leftarrow \emptyset$;
3  $\forall v \in V$ – efficient s-v-paths: $P^*_{sv} \leftarrow \emptyset$ ;
4  $\forall v \in V$ – dimensionality reduced front $\bar{c}_{dr}(P^*_{sv}) \leftarrow \emptyset$;
5  $p_{\text{init}} \leftarrow ()$;
6  $Q \leftarrow Q.\text{insert}(p_{\text{init}})$;
7  while $Q \neq \emptyset$ do
8      $p \leftarrow Q.\text{extractMin}()$ ;
9      $v \leftarrow$ last node of path p. ;                      // If $p = p_{\text{init}}$, $v \leftarrow s$.
10     $\bar{c}_{dr}(P^*_{sv}) \leftarrow \text{merge}\Big(\bar{c}_{dr}(P^*_{sv}), \bar{c}_{dr}(p)\Big)$;  // Without dominance checks in
         Line 4 of merge.
11     Flag success $\leftarrow$ False;
12     if $v \neq t$ then
13         $(Q, \mathcal{NQP}, \text{success}) \leftarrow$ T-propagate$(p, Q, P^*, \mathcal{NQP})$
14     if $v == t$ or success == True then $P^*_{sv}$.append$(p)$;
15     $(p^{new}_v, \mathcal{NQP}) \leftarrow$ T-nextQueuePath$(p, P^*, \mathcal{NQP})$ ;
16     if $p^{new}_v \neq$ NULL then Q.insert$(p^{new}_v)$;
17 return $P_{st}$;
```

PERMANENT PATHS For every $v \in V$, the algorithm stores lists P^*_{sv} of efficient s-v-paths. Note that in contrast to the description of the MDA at the end of the algorithm the permanent s-v-paths for intermediate nodes $v \neq t$ are not requested to be minimal complete sets of efficient paths. For every $v \in V$, permanent s-v-paths are efficient paths but not every efficient s-v-path p is made permanent: if all extensions of p along the outgoing arcs of v are dominated subpaths, p is discarded since it is not part of any efficient s-t-path. In this context it is important to recall that we manage paths as labels. For an s-t-path p whose last arc is (v, t), its label contains the predecessor label that encodes the subpath $p^{s \to v}$. Without storing labels encoding permanent s-u-paths to intermediate nodes u, the T-MDA would not be able to reconstruct the s-t-paths from the set of labels returned by an algorithm's implementation. These labels are not needed anywhere else in the T-MDA which is why if an s-u-path only produces dominated or cost-equivalent extensions along $(u, w) \in \delta^+(u)$, it can be discarded.

DOMINANCE CHECKS Additionally, for every $v \in V$, the algorithm stores the dimensionality reduced front $\bar{c}_{dr}(P_{sv})$ of non-dominated cost vectors belonging to already found efficient s-v-paths. Using Lemma 3.1, the fronts $\bar{c}_{dr}(P_{sv})$ are used to determine if new s-v-paths are dominated. Note that even if an extracted s-v-path p is not made permanent, it is guaranteed to be efficient. Thus, we update the dimensionality reduced front $\bar{c}_{dr}(P_{sv})$ with $\bar{c}(p)$ using a merge operation. By doing so, we get the tightest possible front $\bar{c}_{dr}(P_{sv})$ to discard explored s-v-paths in the future.

Pseudo-Lazy Management of Explored Paths

We now describe the main novelty in the T-MDA. We used the notion of a node's *queue path* already in the description of the MDA. There, for a node $v \in V$, the queue path for v if it exists, is the $\prec_Q$-minimal explored s-v-path that is neither dominated by nor cost-equivalent to a permanent s-v-path. In this chapter, we use the pruning rule described in Section 9.4 and the reduced costs derived in Section 9.3 to compare any explored path with permanent s-t-paths. The following definition puts all conditions for an explored path to be a queue path together.

Definition 9.3 (Queue paths)**.** Consider the permanent s-v- and s-t-paths stored in P^*_{sv} and in P^*_{st} at the beginning of an iteration of the T-MDA. An explored s-v-path p is the *queue path for* v if

1. $\bar{c}(P^*_{sv}) \preceq_D \bar{c}(p)$ does not hold, i.e., p is not dominated by or equivalent to any cost vector in $\bar{c}(P_{sv})$,
2. $c(P^*_{st}) \preceq_D \bar{c}(p)$ does not hold, i.e., p is not dominated by or equivalent to any cost vector in $\bar{c}(P^*_{st})$, and
3. among all explored s-v-paths that fulfill conditions 1. and 2., p is lex. minimal w.r.t. $\bar{c}$.

Explored s-v-paths that are not the queue path for v are stored in so called $\mathcal{NQP}$ (next queue path) lists. The T-MDA maintains an $\mathcal{NQP}_a$ list for every arc $a \in A$ and any explored path that is not a queue path is stored in the $\mathcal{NQP}$ list corresponding to the path's last arc. A high level description of the handling of explored paths in the T-MDA is as follows.

- The priority queue Q and the $\mathcal{NQP}$ lists are sorted in lex. non decreasing order w.r.t. $\bar{c}$.
- For any $v \in V$, there is at most one queue path for v in Q and if there is no queue path for v in Q, all lists $\mathcal{NQP}_a$ for $a \in \delta^-(v)$ are empty.
- The $\mathcal{NQP}$ lists are maintained in a lazy way, i.e., they might contain paths that dominate each other.

- For every $v \in V$, when an s-v-path is extracted from Q, a new queue path for v is picked among the s-v-paths in the lists $\mathcal{NQP}_{uv}$, $(u, v) \in \delta^-(v)$. During this search, explored s-v-paths in these lists that are dominated by or equivalent to paths in P^*_{sv} or P^*_{st} are discarded.

9.5.1 Execution

We proceed with the line-by-line description of the T-MDA. Readers that have read Chapter 4 will find many similarities. However, the usage of the $\mathcal{NQP}$ lists for the new pseudo-lazy management of explored paths requires changes to both subroutines of the MDA.

INITIALIZATION The data structures of the T-MDA are initialized in Lines 1-4. Initially, they are all empty. Before the main loop of the algorithm begins, the trivial path p_{init} from s to itself is inserted into Q (Line 6). The main loop finishes when Q is empty at the beginning of an iteration. Every iteration begins with the extraction of a lex. minimal (w.r.t. $\bar{c}$) path p from Q (Line 8). Since the T-MDA is a label-setting algorithm, paths that are extracted from Q are efficient.

Iterations

The algorithm performs three main tasks in every iteration in addition to the extraction of a $\prec_{\text{lex}}$-minimal path from Q. We assume the existence of containers P^* and $\mathcal{NQP}$ in which the list P^*_{sv} of efficient paths for $v \in V$ and the $\mathcal{NQP}_a$ list for $a \in A$ can be accessed in constant time, respectively.

MERGE OF EXTRACTED PATHS (LINE 10). Let p be an extracted s-v-path for $v \in V$ at the beginning of an iteration. We update the dimensionality reduced front $\bar{c}_{dr}(P^*_{sv})$ (Line 10 of the T-MDA) to enhance future dominance or equivalence checks performed on explored s-v-paths. Since p is efficient, we know that its dimensionality reduced vector of costs needs to be added to $\bar{c}_{dr}(P^*_{sv})$. We thus find the correct insert position for $\bar{c}_{dr}(p) := (\bar{c}_2(p), \dots, \bar{c}_d(p))$ and check if existing vectors in $\bar{c}_{dr}(P^*_{sv})$ are dominated by $\bar{c}_{dr}(p)$. There is no need to check whether $\bar{c}_{dr}(p)$ is dominated. This leads to a speedup in practice.

PROPAGATION OF EXTRACTED PATHS (LINE 13 OF THE T-MDA). The extracted s-v-path p is propagated along the arcs $(v, w) \in \delta^+(v)$ in T-propagate. All references to lines in the following description refer to its pseudocode. For such an arc (v, w), we call the resulting path $q := p \circ (v, w)$ and, as a novelty, we only perform $\preceq_D$-checks using q's reduced costs if q is going to be inserted into Q. I.e. if q does not qualify as a new queue path for w, we postpone its $\preceq_D$-checks. We distinguish three scenarios.

THERE IS NO s-w-PATH IN Q. If $\bar{c}(q)$ is not dominated by and not equivalent to any vector in $\bar{c}_{dr}(P^*_{sw})$ or in $\bar{c}_{dr}(P^*_{st})$, q is inserted into Q and becomes w's queue path (Line 7).

THERE IS AN s-w-PATH $q' \in Q$ AND $\bar{c}(q) \prec_{lex} \bar{c}(q')$. If there is a vector in $\bar{c}_{dr}(P_{sw})$ or in $\bar{c}_{dr}(P_{st})$ that dominates or is equivalent to $\bar{c}(q)$, q is discarded. Otherwise q is inserted into Q via a decreaseKey operation, where it replaces q' (Line 13). If q' is dominated by q, q' is discarded (Line 14). Otherwise assuming that $(x, w) \in A$ is q' last arc, q' is prepended to the list $\mathcal{NQP}_{xw}$ (Line 16). This means that even though q' is no longer in Q, it might re-enter it later.

THERE IS AN s-w-PATH $q' \in Q$ AND $\bar{c}(q') \prec_{lex} \bar{c}(q)$. In this case, q is appended to $\mathcal{NQP}_{vw}$ (Line 20) if it is not dominated by or equivalent to q'. As mentioned earlier, since q is not inserted into Q, no dominance checks w.r.t. to the permanent fronts are made in this case. They are postponed and done in the T-nextQueuePath procedure when needed.

The procedure T-propagate returns the possibly updated priority queue Q and $\mathcal{NQP}$ lists and a success flag whose value is only set to *True* if one of the considered expansions of p is stored either in Q or in an $\mathcal{NQP}$ list. In this case p is added to P^*_{sv}, i.e., p is made permanent since it might be the subpath of an efficient s-t-path p^*.

Algorithm 8: Targeted-Propagate (T-propagate)

```
Input  : s-v-path p, priority queue Q, permanent paths P*, lists of
         explored paths NQP.
Output: Updated priority queue Q and NQP lists and success flag that is
         true only if at least one expansion of p is stored.
1  Flag success ← False;
2  for $w \in \delta^+(v)$ do
3      $q \leftarrow p \circ (v, w)$;
4      if Q does not contain a queue path for w then
5          if not $\bar{c}_{dr}(P^*_{st}) \preceq_D \bar{c}_{dr}(q, \tau_0)$ and not $\bar{c}_{dr}(P^*_{sw}) \preceq_D \bar{c}_{dr}(q, \tau_0)$ then
6              success ← True;
7              Q.insert (q);
8      else
9          $q'$ ← Queue path for w in Q; // Only one s-w-path in Q. Access in
           $\mathcal{O}(1)$.
10         if $\bar{c}(q, \tau_0) \prec_{lex} \bar{c}(q', \tau_0)$ then
11             if not $\bar{c}_{dr}(P^*_{st}) \preceq_D \bar{c}_{dr}(q, \tau_0)$ and not $\bar{c}_{dr}(P^*_{sw}) \preceq_D \bar{c}_{dr}(q, \tau_0)$
                then
12                 success ← True;
13                 Q.decreaseKey(w, q);
14                 if not $\bar{c}(q) \leqslant \bar{c}(q')$ then
15                     $(x, w)$ ← last arc in path $q'$;
16                     Insert $q'$ at the beginning of $\mathcal{NQP}_{xw}$;
17         else
18             if not $\bar{c}(q', \tau 0) \leqslant \bar{c}(q, \tau_0)$ then
19                 success ← True;
20                 Insert q at the end of $\mathcal{NQP}_{vw}$;
21 return $(Q, \mathcal{NQP}, \text{success})$;
```

SEARCH FOR A NEXT QUEUE PATH (LINE 15) OF THE T-MDA. After an s-v-path p is extracted from Q in Line 8 of the T-MDA, there is no s-v-path in

Q. The procedure T-nextQueuePath searches for a new queue path p^* for v according to Definition 9.3 among the explored s-v-paths in the lists $\mathcal{NQP}_{uv}$ for arcs $(u, v) \in \delta^-(v)$.

Algorithm 9: Targeted-NextQueuePath (T-nextQueuePath).

```
Input : s-v-path p, permanent paths P*, lists of explored paths
        NQP.
Output: New queue path for v if one exists, and updated NQP lists.
1  $p^* \leftarrow$ NULL;                                   // Assume $\bar{c}(p^*) = \infty$.
2  for $u \in \delta^-(v)$ do
3      for $p' \in \mathcal{NQP}_{uv}$ do
4          if not $\bar{c}(p') \prec_{lex} \bar{c}(p^*)$ then break;
5          if $\bar{c}_{dr}(P^*_{st}) \preceq_D \bar{c}_{dr}(p')$ or $\bar{c}_{dr}(P^*_{sv}) \preceq_D \bar{c}_{dr}(p')$ then
6              $\mathcal{NQP}_{uv} \leftarrow$ remove $p'$ from $\mathcal{NQP}_{uv}$;
7              continue;
8          else
9              $p^* \leftarrow p'$;
10             break;
11 if $p^* \neq$ NULL then
12     Delete $p^*$ from its $\mathcal{NQP}$ list;  // $p^*$ is the front element in the list.
13 return $(p^*, \mathcal{NQP})$;
```

Initially, p^* is a dummy path and we assume that its cost is $\infty \in \mathbb{R}^d$. The $\mathcal{NQP}$ lists are sorted in lex. non-decreasing order w.r.t. $\bar{c}$ (see Lemma 9.4). Thus, we look for a new value for p^* starting with the first element of each list. Fix an arc (u, v) and consider the list $\mathcal{NQP}_{uv}$. We iterate over $\mathcal{NQP}_{uv}$ and as soon as the iteration finds a path $p' \in \mathcal{NQP}_{uv}$ with $c(p^*) \preceq_{lex} c(p')$ we finish the iteration over $\mathcal{NQP}_{uv}$ (Line 4). Otherwise, the iteration continues until a path that is neither dominated by nor equivalent to paths in P^*_{sv} or in P^*_{st} (Line 5) is found. Hence, the candidate path at which the iteration stops, fulfills Condition 2. and Condition 3. in Definition 9.3. Paths analyzed until the new candidate path in $\mathcal{NQP}_{uv}$ is found, are deleted from the list (Line 6) because, since they are either dominated by or equivalent to permanent s-v or s-t-paths, they are not a candidate to become a queue path. All in all, T-nextQueuePath identifies at most one candidate s-v-path p' with (u, v) as its last arc. When found, p' is guaranteed to have a $\prec_{lex}$-smaller cost vector than p^* and thus, p^* is updated and set to p' (Line 9). Afterwards, the iteration over $\mathcal{NQP}_{uv}$ stops and the outer loop of nextQueuePath continues trying to improve p^* by considering explored paths stored in the $\mathcal{NQP}$ lists corresponding to incoming arcs of v other than (u, v). The path p^* returned by the search is a lex. smallest one among these candidates (Condition 1 in Definition 9.3). Additionally, since p^* is going to be inserted into Q, it is removed from its $\mathcal{NQP}$ list in Line 12. It is easy to see that it is the front element in the list and thus the deletion can be done in $\mathcal{O}(1)$ time. Finally if found, p^* is inserted into Q (Line 16 of Algorithm 7).

Indeed, T-nextQueuePath solves (8) as the routine nextQueuePath does in the MDA. The difference is that instead of building the explored s-v-paths again, the T-MDA stores them in the $\mathcal{NQP}$ lists and can access them directly.

Proposition 9.1. *Consider a node $v \in V$ and assume that the lists $\mathcal{NQP} - a$ for $a \in \delta^-(v)$ are sorted in lex. nondecreasing order. Then, T-nextQueuePath uses target pruning and solves (8). I.e., if the routine returns an s-v-path p^*, the path is a new queue path for v according to Definition 9.3.*

Finally, note that if the extracted path in Line 8 of the T-MDA is an s-t-path, it does not need to be further propagated and it is stored in P^*_{st}. The output of the T-MDA is the list P^*_{st}. The paths can be rebuild using Algorithm 1 if the T-MDA is implemented using labels.

9.5.2 Correctness

We sketch the correctness of the T-MDA because it is easy to see that the correctness proof is similar to the correctness proof of the MDA. The statements in this section analyze the interplay between the $\mathcal{NQP}$ lists and the priority queue Q to prove that paths are extracted in lexicographic order from Q. They lead to Lemma 9.5 that guarantees that paths are extracted from Q in lex. non-decreasing order. The statement is equivalent to Lemma 4.4 for the MDA. Having proven this statement, the correctness of the T-MDA is proven as the correctness of the MDA. For a complete self-contained correctness proof of the T-MDA we point the reader to (Maristany de las Casas, Kraus, et al., 2023).

In Proposition 9.1 we assumed that the $\mathcal{NQP}$ lists are sorted throughout the T-MDA. We need to prove that the assumption is true. They clearly are sorted as long as they only contain at most one path. Whether they remain sorted once they start storing more than one path depends on the extraction order of paths from Q. We first prove in Lemma 9.3 that while the $\mathcal{NQP}$ lists contain at most one element, paths are extracted from Q in lex. nondecreasing order w.r.t. $\bar{c}$. With this knowledge, we then prove inductively in Lemma 9.4 that the constant time insertion of paths at the beginning or at the end of the $\mathcal{NQP}$ lists performed in T-propagate do always respect the ordering of the lists.

Lemma 9.3. *Assume the T-MDA is used to solve a One-to-One MOSP instance in which the cardinality of every $\mathcal{NQP}_a$ list, $a \in A$, is at most one during the whole algorithm's execution. Then, paths are extracted from Q in Line 8 of T-MDA in lex. nondecreasing order.*

Proof. Consider the arc $a = (u, v) \in A$. An explored s-v-path q with a as its last arc is stored in $\mathcal{NQP}_a$ if there is a lex. smaller queue path q' for v in Q when q is explored. When q' or another s-v-path is extracted, T-nextQueuePath finds a new queue path p^* for v in the lists $\mathcal{NQP}_{a'}$ for $a' \in \delta^-(v)$. Since all $\mathcal{NQP}$ lists are sorted because they contain at most one element, the final value of p^* in T-nextQueuePath is correct.

This path is guaranteed to be lex. greater than the extracted path. The remainder of the formal proof, while heavy on notation, is trivial. □

Now, we need to generalize this observation to the case in which the $\mathcal{NQP}$ lists contain more than a single element. During the solution process of any One-to-One MOSP instance using the T-MDA we can identify the first iteration in which the cardinality of an $\mathcal{NQP}$ list becomes greater than one. By Lemma 9.3 we know that until this iteration, paths are extracted from Q in lex. non-decreasing order. For this to keep holding, we need to prove that the $\mathcal{NQP}$ lists remain sorted correctly using the prepend and append operations in Line 16 and in Line 20 of T-propagate. This is the crux of our pseudo-lazy management of explored paths: by using constant time insertions into a list of explored paths, we can keep the list sorted.

Lemma 9.4. *Every insertion of paths at the beginning or at the end of an $\mathcal{NQP}$ list leaves the list ordered in lex. nondecreasing order. Moreover, for any node $w \in V$ the s-w-path in Q is not lex. greater than any path in the $\mathcal{NQP}_a$ lists for $a \in \delta^-(w)$.*

Proof. Every line referenced in this proof refers to T-propagate. Assume that the T-MDA calls the routine T-propagate and for the first time, an $\mathcal{NQP}$ list becomes a second path added to it. Using the notation from the pseudocode, two s-w-paths are involved: the queue path q' for w and the new explored s-w-path q. We know from Lemma 9.3 and its proof that q' is lex. smaller than any path stored in an $\mathcal{NQP}_a$ list for $a \in \delta^-(w)$.

The first option is that q is lex. smaller than q' (Line 10) and q passes the $\preceq_D$-checks in Line 5. Then, q replaces q' in Q (Line 13). If additionally q' is not dominated by or equivalent to q (Line 14) it is added at the beginning of $\mathcal{NQP}_{xw}$ (Line 16), where (x, w) is the last arc of q'. Since q' is lex. smaller than every path in an $\mathcal{NQP}_a$ list for $a \in \delta^-(w)$, adding it at the beginning of $\mathcal{NQP}_{xw}$ leaves the list sorted correctly.

The second possibility is that q' is lex. greater than q. Then if q' does not dominate q and both paths are not equivalent (Line 18), q is inserted at the end of $\mathcal{NQP}_a$ (Line 20), where $a = (v, w)$ is the last arc of q. Let q'' be the path that is already in $\mathcal{NQP}_a$. It is stored in this list because a is its last arc. Thus, q'' was explored after expanding an efficient s-v-path p'' during the call to T-propagate in an earlier iteration. By Lemma 9.3 we know that so far paths have been extracted from Q in lex. nondecreasing order. Thus, for the s-v-path p extracted from Q in the current iteration, we have $\bar{c}(p'') \prec_{lex} \bar{c}(p)$ and consequently

$$\bar{c}(p'') + \bar{c}(a) \prec_{lex} \bar{c}(p) + \bar{c}(a).$$

Thus, inserting q at the end of $\mathcal{NQP}_a$ leaves the list sorted.

We can repeat the same argument inductively to prove the statement for any cardinality of an $\mathcal{NQP}$ list. Note that in both cases, the s-w-path in Q after the call to T-propagate is a lex. smallest explored s-w-path. This directly implies that the s-w-path in Q is a queue path for w according to Definition 9.3. □

The following lemma is the T-MDA equivalent to Lemma 4.4 in the correctness proof of the MDA.

Lemma 9.5. *Let p and q be two paths extracted from Q in Line 8 of the T-MDA. If p is extracted in an earlier iteration than q, then $p \preceq_{lex} q$.*

Proof. We prove the statement inductively and see Lemma 9.3 and its proof as the induction basis. Following Lemma 9.4 the paths in Q are queue paths according to Definition 9.3. Let p be an s-v-path extracted from Q at the beginning of an iteration of the T-MDA. Given Proposition 9.1 and Lemma 9.4

we can use Proposition 9.1 to conclude that T-nextQueuePath solves (8) correctly. I.e., the path added to Q in Line 16 of the T-MDA is a new queue path for v. The remainder of the proof is analogous to the proof of Lemma 4.4. $\square$

Since the lexicographic order used to sort paths in Q is compatible with the dominance order (Definition 3.5), an extracted s-v-path for any $v \in V$ at the beginning of an iteration of the T-MDA is neither dominated by nor equivalent to paths already in P^*_{sv} or in P^*_{st} or added later to these lists. This same statement is proven for the MDA in Lemma 4.3. The remaining two statements leading to the correctness of the MDA, namely Lemma 4.5 and Theorem 4.1 can be equivalently stated and proven for the T-MDA using Lemma 9.5. Thus, we fast-forward to the correctness result for the T-MDA.

Theorem 9.2 (Correctness of the T-MDA). *The T-MDA returns a minimal complete set P^*_{st} of efficient s-t-paths.*

9.6 EFFICIENCY OF THE T-MDA

It is easy to see that the asymptotic running time bound of the T-MDA coincides with the bound (11) of the MDA. Even if more explored paths are stored in the $\mathcal{NQP}$ lists, we only prepend or append paths to these lists and only extract paths from the beginning the lists. Thus, this extra constant-time effort per path can be neglected since the per-path effort in the T-MDA is still asymptotically dominated by the $\preceq_D$-checks. The memory consumption bound of the T-MDA coincides with the bound (14) of Martins's algorithm since both paths store explored paths explicitly.

Consider a One-to-One MOSP instance that is solved with the T-MDA and with the MDA. Assume the used heuristic is $\pi(v) = 0$ for every $v \in V$ such that $\bar{c} = c$. Moreover, let the MDA also use target pruning checks $c(P^*_{st}) \preceq_D c(p)$ for explored paths p. In this situation both algorithms need the same iterations to output P^*_{st}. It turns out however, that our design choices for the T-MDA make the algorithm approximately twice as efficient as the MDA with regard to the iterations both algorithms perform per second.

There are three main reasons for the superior efficiency of the T-MDA.

- In T-nextQueuePath the queue path candidates do not need to be rebuilt since they are stored in their corresponding $\mathcal{NQP}$ lists. Let $q = p \circ (u, v)$ be such a path. In the MDA, in Line 5 of nextQueuePath, the permanent label representing the s-u-path p needs to be accessed and expanded along $a = (u, v)$ which entails the computation of the sum $c(p) + c(a)$. If millions of paths are considered, the $\mathcal{O}(d)$ effort for these repeated sums is not negligible.

- Let p be an s-v-path that is made permanent in both algorithms. Assume that for an arc $(v, w) \in \delta^+(v)$, the new explored path $q = p \circ (v, w)$ is inserted into Q directly after it is built in propagate and in T-propagate. Additionally, assume that in both algorithms q remains the queue path for w until it is extracted and made permanent. In the T-MDA q is not stored in any $\mathcal{NQP}$ list and thus only one set of

$\preceq_D$-checks (w.r.t. P^*_{sv} and P^*_{st}) is performed to assess the relevance of q. In the MDA however, p is stored in P^*_{sv} and during a call to nextQueuePath, the $\mathrm{lastProcessedPath}[v, w]$ index must pass p's position in P^*_{sv}. When this happens, p is expanded again along (v, w) and the dominance or equivalence of q is assessed again. Hence, for every path that is made permanent without being replaced in Q after its insertion in propagate or in T-propagate, the MDA does two sets of $\preceq_D$-checks and the T-MDA only one. In practice, most permanent paths follow this route to acquire their permanent status.

- Since explored paths that are not queue paths are stored in $\mathcal{NQP}$ lists in the T-MDA, there is no need to store the labels encoding permanent paths with the paths' cost vectors. Thus, the T-MDA can take advantage of less memory consuming permanent labels in practice as described in Remark 6.1 for Martins's algorithm.

9.7 NAMOA$^*_{\mathrm{DR}}$-LAZY ALGORITHM

The NAMOA$^*_{\mathrm{dr}}$ algorithm was introduced in (Pulido et al., 2015). It is an A*-based algorithm for One-to-One MOSP problems. It equips the original NAMOA* algorithm from (Pulido et al., 2014) with the dimensionality reduction technique discussed in Section 3.5. Similarly to the T-MDA, the algorithm uses a priority queue Q of paths. In the NAMOA$^*_{\mathrm{dr}}$ algorithm, the coexistence of multiple paths in Q with same end nodes is allowed as long as they do not dominate each other. To ensure this invariant, the NAMOA$^*_{\mathrm{dr}}$ algorithm uses merge operations (cf. Definition 3.7) after exploring new paths and checking that they are not dominated by and not equivalent to any permanent path. The computational results in (Maristany de las Casas, Sedeño-Noda, & Borndörfer, 2021) substantiate that in label-setting MOSP algorithms, merge operations during the expansions of paths should be avoided to obtain competitive running times.

Inspired by the label-setting BOSP algorithm presented in (Ahmadi et al., 2021) that avoids merge operations, we introduce the *NAMOA$^*_{dr}$-lazy* algorithm. The pseudocode is given in Algorithm 10 and uses the same notation as our description of the T-MDA. Explored paths are handled in a lazy way. As explained in this chapter's introduction, this means that every new explored path that is not dominated by or equivalent to existing permanent paths (Line 14) when it is analyzed for the first time is inserted into the algorithm's priority queue (Line 17). Hence, the queue can contain exponentially many paths simultaneously and for any $v \in V$, s-v-paths in the queue can dominate each other. As a consequence and in contrast to the invariant regarding the efficiency of extracted paths in the T-MDA, paths that are extracted from the queue (Line 6) are not guaranteed to be efficient. To avoid the storage and expansion of such paths, the algorithm needs to check if they are dominated right after their extraction. This is achieved as a side-effect of the call to the merge procedure to update the dimensionality reduced fronts in Line 8. Assume the s-v-path p is extracted from Q. A merge operation possibly updates the front $\bar{c}_{dr}(P_{sv})$ by inserting $\bar{c}_{dr}(p)$ and deleting elements dominated by $\bar{c}_{dr}(p)$. However if $\bar{c}_{dr}(p)$ is not added to the front, p is dominated (cf. Lemma 3.1).

The correctness of the NAMOA$^*_{dr}$-lazy can be proven following the correctness proof of the BOSP algorithm in (2021). We benchmark NAMOA$^*_{dr}$-lazy against the T-MDA in Section 9.8.

Algorithm 10: NAMOA$^*_{dr}$-lazy

```
Input  : MOSP instance $\mathcal{I} = (G, s, t, d, c)$, heuristic $\pi$ for $\mathcal{I}$.
Output : Minimal complete set $P^*_{st}$ of efficient s-t-paths.
1  Priority queue of paths $Q \leftarrow \emptyset$;          // Sorted according to $\bar{c}$.
2  $\forall v \in V$ – efficient s-v-paths: $P^*_{sv} \leftarrow \emptyset$ ;
3  $p_{\text{init}} \leftarrow ()$;
4  $Q \leftarrow Q.\text{insert}(p_{\text{init}})$;
5  while $Q \neq \emptyset$ do
6      $p \leftarrow Q.\text{extractMin}()$ ;
7      $v \leftarrow$ last node of path p. ;                   // If $p = p_{\text{init}}$, $v \leftarrow s$.
8      $\bar{c}_{dr}(P^*_{sv}) \leftarrow \text{merge}\Big(\bar{c}_{dr}(P^*_{sv}), \bar{c}_{dr}(p)\Big)$;
9      if $\bar{c}_{dr}(p) \notin \bar{c}_{dr}(P^*_{sv})$ then continue;
10     Flag success $\leftarrow$ False;
11     if $v \neq t$ then
12         for $w \in \delta^+(()v)$ do
13             $p^{new}_w \leftarrow p \circ (v, w)$;
14             if $c_{dr}(P^*_{st}) \preceq_D \bar{c}_{dr}(p^{new}_w)$ or $\bar{c}_{dr}(P^*_{sw}) \preceq_D \bar{c}_{dr}(p^{new}_w)$
                then
15                 continue
16             else
17                 Q.insert $(p^{new}_w)$;
18                 success $\leftarrow$ True;
19     if $v == t$ or success == True then $P^*_{sv}$.append(p);
20 return $P^*_{st}$;
```

9.8 EXPERIMENTS

We compare the performance of the T-MDA and the NAMOA$^*_{dr}$-lazy algorithm on 3-dimensional One-to-One MOSP instances. We decided not to include the original NAMOA$^*_{dr}$ algorithm from (Pulido et al., 2015) in this chapter. In Figure 16 we show the results of the direct comparison of the new NAMOA$^*_{dr}$-lazy algorithm and our implementation of the original NAMOA$^*_{dr}$ algorithm on the road network of New York. Already on this network (the smallest considered road network) it becomes apparent that the NAMOA$^*_{dr}$-lazy algorithm performs better consistently. The reason is that the NAMOA$^*_{dr}$ algorithm conducts merge operations after every successful expansion of a path along one of its outgoing arcs. From the experiments in Chapter 6 we know that merge operations should be avoided to enhance the practical performance of MOSP algorithms in practice. On the instances

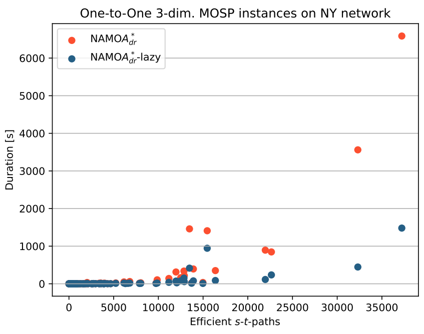


Figure 16: Comparison of our implementation of the original NAMOA$^*_{\text{dr}}$ algorithm and the new NAMOA$^*_{\text{dr}}$-lazy algorithm on 100 3-dimensional One-to-One MOSP instance defined on the NY network.

shown in Figure 16 the NAMOA$^*_{\text{dr}}$-lazy algorithm is $\times 1.96$ times faster on average. If we consider only the instances that output more than 1000 s-t-paths, the new algorithm is already $\times 2.51$ times faster on average.

SETUP We use the 3-dimensional One-to-One MOSP instances defined on EXP graphs, on NetMaker graphs, and on road networks that we already described in Section 6.2.1. We calculate the heuristics π as described in Section 9.3 and pass them to both algorithms as part of their input. Thus, we do not report preprocessing times in this section. Further data structures coincide with the ones described in Section 6.2. Only the $\mathcal{NQP}$ lists for the T-MDA are new in this section. They are implemented as singly linked lists since we only need to insert at the beginning and at the end and we only remove elements from the beginning of the lists. The used hardware and the time and memory limits are identical to the ones described in Section 6.3 for the comparison of the MDA and Martins's algorithm. The code, the detailed statistics collected for every instance, the scripts to generate the tables and the plots in this section, and the plots that are not implicitly included here are available in (Maristany de las Casas, 2023b).

9.8.1 Results

The results of the experiments are summarized in Table 8, Table 9, and Table 11. Throughout this section we use scatter plots to visualize the results. We plot the instances' running time depending on the cardinality of the output sets P^*_{st}. Again, all averages are geometric means. The handling of instances that are solved only by the T-MDA coincides with the description in Remark 6.2. There are no instances solved only by the NAMOA$^*_{\text{dr}}$-lazy algorithm. Given the coincident experimental setup and evaluation scripts, the results in this section are comparable to those in Section 6.3.

Exponential Instances

In Table 8 we present the results obtained from the EXP instances. The instances defined on the graphs EXP3-EXP13 are left out because both algorithms solve them in less than 0.1 milliseconds. Instances from the EXP63

			time		
Graph	N	N_t	T-MDA	NAMOA$^*_{dr}$-lazy	Speedup
EXP15	382	128	0.0002	0.0002	1.00
EXP17	766	256	0.0001	0.0002	2.00
EXP19	1534	512	0.0006	0.0003	0.50
EXP21	3070	1024	0.0008	0.0007	0.87
EXP23	6142	2048	0.0012	0.0028	2.33
EXP25	12 286	4096	0.0025	0.0036	1.44
EXP27	24 574	8192	0.0053	0.0118	2.23
EXP29	49 150	16 384	0.0101	0.0222	2.20
EXP31	98 302	32 768	0.0174	0.0238	1.37
EXP33	196 606	65 536	0.0403	0.0526	1.31
EXP35	393 214	131 072	0.0487	0.0861	1.77
EXP37	786 430	262 144	0.0959	0.1581	1.65
EXP39	1 572 862	524 288	0.1599	0.3187	1.99
EXP41	3 145 726	1 048 576	0.2849	0.7237	2.54
EXP43	6 291 454	2 097 152	0.5639	1.6826	2.98
EXP45	12 582 910	4 194 304	1.1469	3.1570	2.75
EXP47	25 165 822	8 388 608	2.1569	6.4171	2.98
EXP49	50 331 646	16 777 216	4.1345	13.2804	3.21
EXP51	100 663 294	33 554 432	7.8872	27.3165	3.46
EXP53	201 326 590	67 108 864	15.8135	57.2326	3.62
EXP55	402 653 182	134 217 728	32.5927	115.7366	3.55
EXP57	805 306 366	268 435 456	60.3624	240.8411	3.99
EXP59	1 610 612 734	536 870 912	122.2462	500.0935	4.09
EXP61	3 221 225 470	1 073 741 824	242.3084	1037.1834	4.28
EXP63	6 442 450 942	2 147 483 648	480.8830	-	-

Table 8: NAMOA$^*_{dr}$-lazy vs. T-MDA on the EXP instance set.

onward are not solvable by any of the algorithms because of issues in our implementations indexing vectors with more than 2^{32} entries. This is because, to allow fast cache access to labels, we assume that a the maximum size of a minimal complete set of efficient paths is at most 2^{32}. This assumption is reasonable for every type of instances other than EXP instances. The reason why the instance EXP63 can be solved by the T-MDA in this chapter and is not solved in Chapter 6 by the MDA is the memory saving storage of permanent paths explained in Remark 6.1.

We observe that the T-MDA outperforms the NAMOA$^*_{dr}$-lazy algorithm consistently and the speedup increases as the graphs' size increases. The biggest instance solved by the T-MDA, the EXP63 instance, is not solved by the NAMOA$^*_{dr}$-lazy algorithm. Thus, the biggest speedup, $\times 4.28$, is reached for the EXP61 instance.

We see the T-MDA as a tuned version of the MDA for performance in practice. Similarly, the NAMOA$^*_{dr}$-lazy algorithm is related to Martins's algorithm. It is remarkable that in this chapter, the speedups in favor of the T-MDA are drastically smaller than speedups on the EXP instances in favor of the MDA in Chapter 6. In other words, while the T-MDA is consistently faster than the MDA ($\times 1.59$ on the EXP61 instance), the running time im-

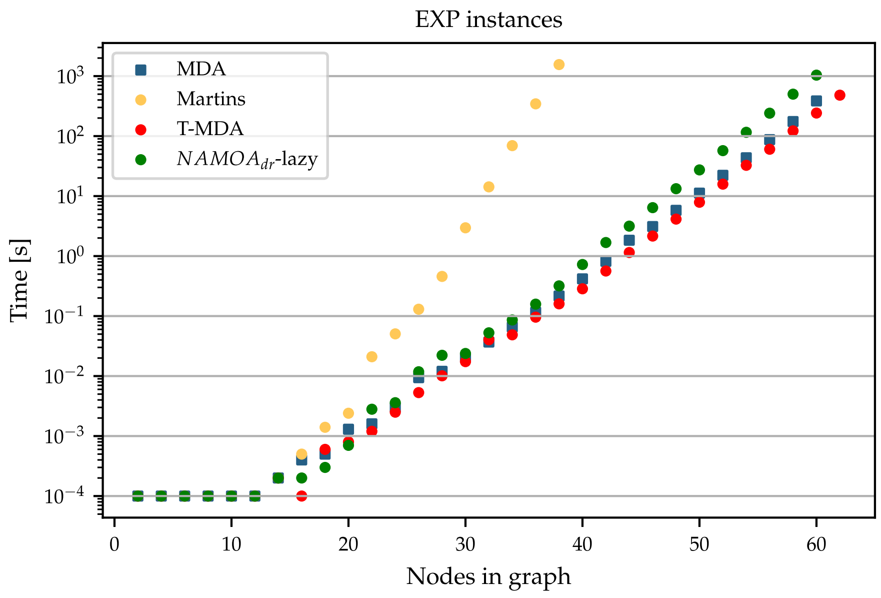


Figure 17: Running times of the MDA, Martins's algorithm, the T-MDA, and the NAMOA$^*_{\mathrm{dr}}$-lazy algorithm on 3d EXP instances.

provement achieved by the NAMOA$^*_{\mathrm{dr}}$-lazy algorithm w.r.t. Martins's algorithm ($\times$5022.13 on the EXP39 instance) is much higher. The reason is that the lazy path management in the NAMOA$^*_{\mathrm{dr}}$-lazy algorithm gets rid of the merge operations needed in Martins's algorithm. Figure 17 shows a comparison of the four algorithms and is a good indicator of what to expect from the remaining results reported in this chapter: the T-MDA is faster than the NAMOA$^*_{\mathrm{dr}}$-lazy algorithm but the difference is not as pronounced as in Chapter 6. The comparison in the remainder of this chapter thus boils down to the comparison of the two different lazy management approaches used in the T-MDA and in the NAMOA$^*_{\mathrm{dr}}$-lazy algorithm. This comparison is novel since none of these techniques was published prior to (Maristany de las Casas, Kraus, et al., 2023) in a general multiobjective context.

General Observations for NetMaker and Road Instances

The columns *Extr. avg.* in Table 9 and in Table 11 report the average number of extractions from the heap. In the T-MDA this number coincides with the number of iterations. The NAMOA$^*_{\mathrm{dr}}$-lazy algorithm cannot guarantee that extracted paths are efficient and thus, the extracted paths can be discarded in Line 8 of Algorithm 10. Thus, the difference between the values reported in the *Extr. avg.* columns of both algorithms unveils the number of unneeded insertions into the heap performed by the NAMOA$^*_{\mathrm{dr}}$-lazy algorithm. On the T-MDA side, the handling of the $\mathcal{NQP}$ lists and the calls to nextQueuePath* stand in contrast to these unneeded insertions. On both types of instances we observe that the NAMOA$^*_{\mathrm{dr}}$-lazy algorithm is faster than the T-MDA on instances with a small output. Such instances require few iterations to be solved. Hence, even if the NAMOA$^*_{\mathrm{dr}}$-lazy algorithm inserts more paths than the T-MDA algorithm into the heap, the heap can be handled efficiently. In this situation, maintaining the $\mathcal{NQP}$ lists in the T-MDA turns out to be less efficient than inserting every relevant explored path into the heap. As the size of the output sets P^*_{st} increases, instances are solved faster by the T-MDA. Here, keeping the size of the heap bounded by holding back explored paths in the $\mathcal{NQP}$ lists of the T-MDA to possibly discard them without ever entering the queue turns out to be a consistent advantage. On NetMaker instances with an output set of more than 1000 paths, the T-MDA is $\times 1.29$ to $\times 1.87$ times faster than the NAMOA$^*_{dr}$-lazy algorithm. On road networks, the T-MDA is $\times 1.02$ to $\times 1.33$ times faster on instances with an output set containing more than 5000 paths. In the next two sections we discuss the results in more detail. For every netM-n group of graphs and for every road network, we partition the solved instances into intervals depending on the cardinality $|P^*_{st}|$ of the output sets to obtain a better insight into the asymptotic behavior of the algorithms' running times.

NetMaker Graphs

The NetMaker results are summarized in Table 9. All instances were solved within the time limit. Figure 18 and Figure 19 are two plots showing the performance of both algorithms on the netM-15 and netM-30 instances. For all other netM-n instances' groups the equivalent plots are contained in the results folder in (Maristany de las Casas, 2023b).

In every netM-n group we observe that the the T-MDA performs better when the number of computed efficient paths grows. Already on the instances with 100-1000 output paths the T-MDA is consistently faster. The speedups in every output size interval remain stable among the different netM-n groups. This indicates that both algorithm's effort is decoupled from the underlying graph and focuses on the handling of explored and efficient paths. That is precisely what we aimed to benchmark in this section. For instances with 500-1000 output s-t-paths, the T-MDA is approximately twice as fast as the NAMOA$^*_{dr}$-lazy algorithm. It reaches an average speedup factor of 2.18 on netM-25 instances. There is only one instance in the netM-30 group whose output contains more than 10000 s-t-paths. Interestingly, this instance is solved very fast by both algorithms. This instance can be considered an outlier.

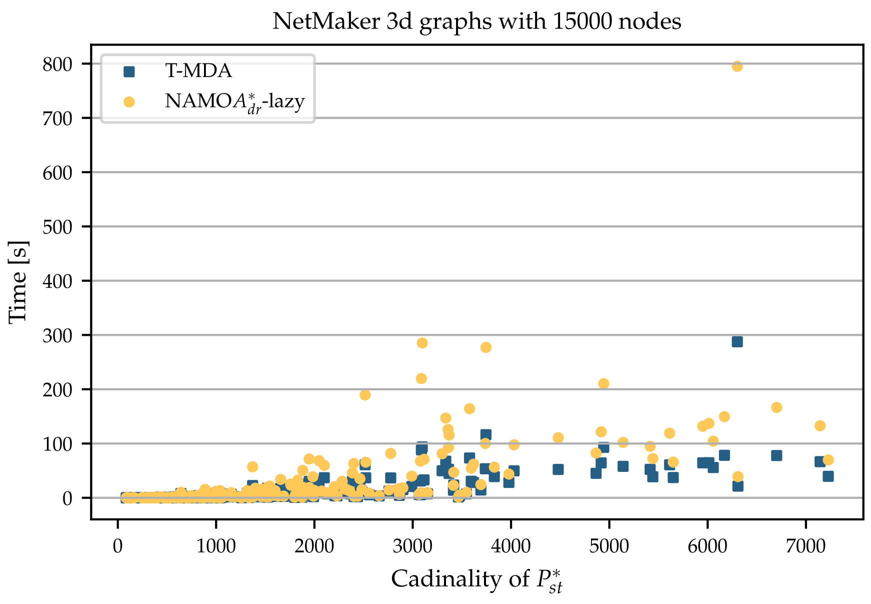


Figure 18: Running times of the NAMOA$^*_{\text{dr}}$-lazy algorithm and the T-MDA on netM-15 networks.

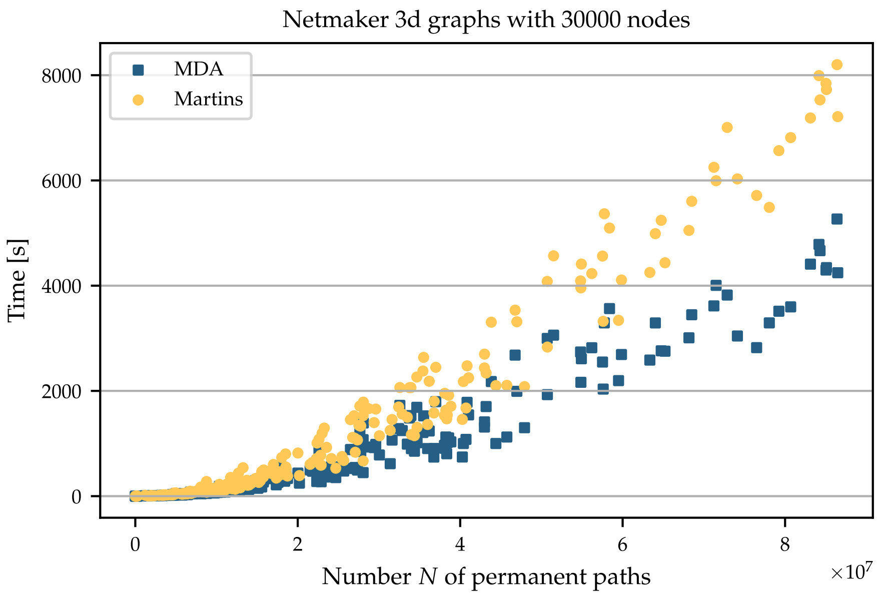


Figure 19: Running times of the NAMOA$^*_{\text{dr}}$-lazy algorithm and the T-MDA on netM-30 networks.

	$\lvert P^*_{st}\rvert$ in range	Instances	$\lvert P^*_{st}\rvert$ avg.	T-MDA		NAMOA$^*_{dr}$ lazy		Speedup
				Extr. avg.	Time avg.	Extr. avg.	Time avg.	
netM-5	$(0, 100]$	11	22.79	462.73	0.0005	802.69	0.0004	0.88
	$(100, 1000]$	128	434.37	94610.10	0.1263	200643.00	0.1278	1.01
	$(1000, 5000]$	95	1702.80	681439.10	2.5260	1935252.96	3.2497	1.29
	$(5000, 10000]$	6	5918.56	1399164.78	8.5378	4476651.71	12.7678	1.50
netM-10	$(0, 100]$	4	39.61	975.83	0.0008	1652.07	0.0007	0.94
	$(100, 1000]$	106	440.32	113964.42	0.1451	227397.59	0.1463	1.01
	$(1000, 5000]$	102	2043.94	1279011.86	5.3592	3390431.52	7.5666	1.41
	$(5000, 10000]$	8	5518.99	4706012.42	35.3466	12873353.15	63.5320	1.80
netM-15	$(0, 100]$	5	91.08	11395.57	0.0080	24028.05	0.0085	1.07
	$(100, 1000]$	98	475.27	160767.82	0.2343	329351.02	0.2452	1.05
	$(1000, 5000]$	123	2048.25	1724092.58	8.1998	4767548.66	12.0421	1.47
	$(5000, 10000]$	14	6051.60	6889085.20	58.9873	19766597.08	115.3208	1.96
netM-20	$(0, 100]$	3	72.07	4433.48	0.0032	8532.91	0.0029	0.92
	$(100, 1000]$	91	479.66	218057.42	0.3511	434363.35	0.3486	0.99
	$(1000, 5000]$	131	2167.24	2101620.29	10.2214	5757119.51	15.0828	1.48
	$(5000, 10000]$	15	6223.22	7639237.35	65.4018	23667406.25	131.2074	2.01
netM-25	$(0, 100]$	4	27.18	727.09	0.0005	1225.56	0.0005	1.09
	$(100, 1000]$	89	503.36	233997.17	0.3815	463089.62	0.3813	1.00
	$(1000, 5000]$	93	2231.47	3352843.49	19.0885	9451181.49	31.0526	1.63
	$(5000, 10000]$	14	5788.98	9442038.02	87.4058	30470870.89	190.6421	2.18
netM-30	$(0, 100]$	2	41.57	843.65	0.0009	2486.59	0.0008	0.89
	$(100, 1000]$	90	490.07	232932.15	0.3827	465534.24	0.3892	1.02
	$(1000, 5000]$	125	2206.32	2810058.13	14.5021	7603577.00	22.1199	1.53
	$(5000, 10000]$	22	6202.14	11239407.94	105.1081	33657201.87	219.3367	2.09
	$(10000, 11566]$	1	11 566.00	1346554.00	8.0691	4173220.00	11.0535	1.37

Table 9: T-MDA vs. NAMOA$^*_{dr}$-lazy algorithm on NetMaker graphs. Instances are separated depending on the cardinality of the output sets P^*_{st}. All averages are geometric means.

Road Network	MDA	T-MDA
NY	73	100
BAY	65	97
COL	46	92
FLA	17	63
NE	13	69
LKS	8	25
E	–	36
W	–	13
CTR	–	8

Table 10: Solvability of the 100 s-t-pairs defined on every road network.

Road Networks

The results collected on road networks are summarized in Table 11. Figure 20 and Figure 21 are two plots containing the results in COL and FLA, respectively. For every other road network the corresponding plot is stored in the results folder in (Maristany de las Casas, 2023b).

Even if the algorithms in this chapter are faster than the MDA and Martins's algorithm in Chapter 6, not all instances are solved within the time limit. In Table 10 we can compare how many instances are solved by the MDA and the T-MDA in every road network. The T-MDA and the NAMOA$^*_{\text{dr}}$-lazy algorithms solve instances defined on the W, E, and CTR road networks for the first time. On these four large networks, we are facing instances with more than 75000 efficient s-t-paths.

In Table 11 we observe that, as noted in Section 9.8.1, the speedups favor the T-MDA more as the output size grows. However, the greatest speedups of around $\times 1.3$ are smaller than the ones achieved on NetMaker instances. The reason, again, is the correlation of the two first arc cost dimensions. In our encoding, the first arc cost dimension is the arc's length and the second one the duration needed to traverse the arc. According to Lemma 9.5 that also holds for the NAMOA$^*_{\text{dr}}$-lazy algorithm, paths are explored with increasing overall length. The correlation of these arc costs with the duration implies that the found lex. minimal solutions (minimal first cost dimension) also have a *very good* second cost dimension, i.e., an almost minimal overall duration. Hence, regarding these two cost dimensions, the NAMOA$^*_{\text{dr}}$-lazy algorithm does not perform many more (erroneous) heap-insertions compared to the T-MDA algorithm. As a representative example, looking at the instances in the netM-25 group whose output contain between 5000 and 10000 paths, the division of the values in the *Extr. Avg.* columns of the T-MDA by the values in the *Extr. Avg.* columns of the NAMOA$^*_{\text{dr}}$-lazy algorithm gives a ratio of approximately 0.31. The same calculation on the COL instances whose output contain between 5000 and 10000 paths gives a ratio of approximately 0.88. Hence, on road instances the T-MDA is holding back paths in the $\mathcal{NQP}$ before their insertion into the heap but almost every path that is held back ends up being inserted into the heap. Despite this fact, as shown also in Figure 20 and Figure 21, the superiority of the T-MDA on these instances is consistent.

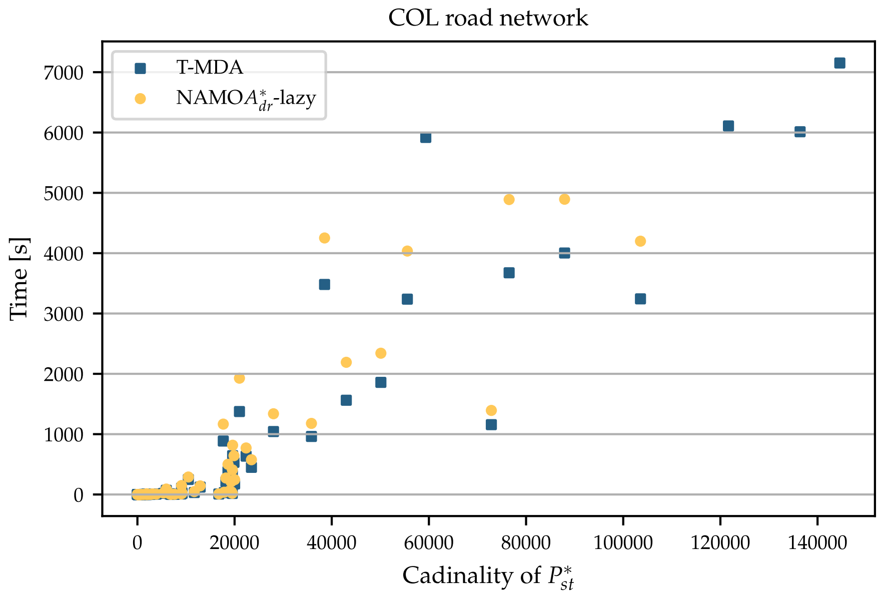


Figure 20: Running times of the NAMOA$^*_{\text{dr}}$-lazy algorithm and the T-MDA on 82 COL instances.

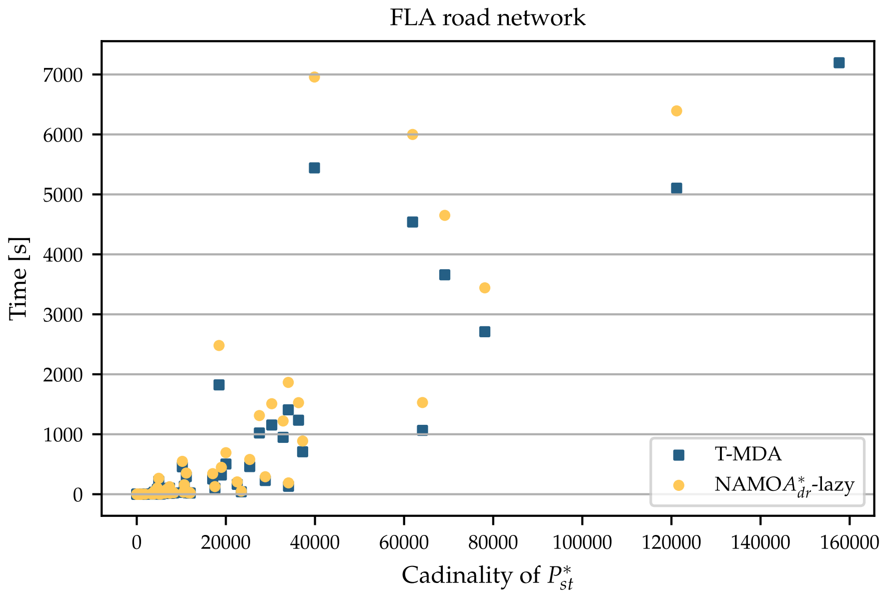


Figure 21: Running times of the NAMOA$^*_{\text{dr}}$-lazy algorithm and the T-MDA on 64 FLA instances.

Table 11: T-MDA vs. NAMOA$^*_{dr}$-lazy algorithm on road networks. Instances are separated depending on the cardinality of the output sets P^*_{st}. All averages are geometric means.

		T-MDA				NAMOA$^*_{dr}$ lazy				Speedup
	$\lvert P^*_{st}\rvert$ interval	Solved	Extr. Avg.	$\lvert P^*_{st}\rvert$ Avg.	Time Avg.	Solved	Extr. Avg.	$\lvert P^*_{st}\rvert$ Avg.	Time Avg.	
NY	$(0, 100]$	15	1661	22	<0.0001	15	1881	22	<0.0001	-
	$(100, 1000]$	31	64 442	328	0.0264	31	79 846	328	0.0222	0.84
	$(1000, 5000]$	29	1 744 416	2699	1.6816	29	2 239 196	2699	1.8716	1.11
	$(5000, 10000]$	9	6 939 950	7253	10.7663	9	8 934 772	7253	14.3497	1.33
	$(10000, 50000]$	16	31 336 640	16 057	96.6909	16	39 950 499	16 057	127.4488	1.32
BAY	$(0, 100]$	9	2085	21	0.0007	9	2283	21	0.0005	0.70
	$(100, 1000]$	28	109 122	324	0.0481	28	126 307	324	0.0403	0.84
	$(1000, 5000]$	26	1 764 644	2252	1.7948	26	2 087 324	2252	1.8562	1.03
	$(5000, 10000]$	11	10 792 037	7010	26.1110	11	12 786 645	7010	31.2632	1.20
	$(10000, 50000]$	20	63 375 493	20 787	375.4029	20	73 988 871	20 787	464.2076	1.24
	$(50000, 61884.0]$	3	352 444 482	56 894	4710.1434	3	405 013 140	56 894	6025.2749	1.28
COL	$(0, 100]$	12	2028	23	<0.0001	12	2279	23	<0.0001	-
	$(100, 1000]$	16	162 076	467	0.0706	16	191 104	467	0.0612	0.87
	$(1000, 5000]$	21	1 316 319	1893	1.2967	21	1 519 834	1893	1.3297	1.03
	$(5000, 10000]$	10	8 028 917	7444	14.1767	10	9 055 786	7444	16.8215	1.19
	$(10000, 50000]$	23	45 844 366	20 234	235.9701	23	52 309 268	20 234	298.0236	1.26
	$(50000, 144586.0]$	10	385 066 841	85 190	3726.5321	6	350 972 008	72 173	4519.2881	1.21
FLA	$(0, 100]$	6	1218	21	<0.0001	6	1343	21	<0.0001	-
	$(100, 1000]$	8	99 836	393	0.0423	8	121 278	393	0.0383	0.90
	$(1000, 5000]$	10	6 593 880	3400	12.6243	10	7 839 828	3400	14.2287	1.13
	$(5000, 10000]$	9	8 328 965	6406	19.6387	9	9 791 709	6406	22.6715	1.15
	$(10000, 50000]$	24	46 566 424	20 456	273.2398	24	54 550 474	20 456	346.4100	1.27

	(50000, 157640.0]	6	327 646 620	86 174	3474.7360	5	335 678 694	76 369	4348.0721	1.25
NE	(0, 100]	1	2220	34	0.0007	1	2394	34	0.0005	0.71
	(100, 1000]	7	82 474	337	0.0365	7	95 508	337	0.0296	0.81
	(1000, 5000]	13	2 723 805	2936	3.2496	13	3 233 255	2936	3.3697	1.04
	(5000, 10000]	9	8 340 508	6321	18.3182	9	9 814 359	6321	21.1597	1.16
	(10000, 50000]	23	83 710 454	24 838	408.5482	23	100 983 553	24 838	515.9635	1.26
	(50000, 158843.0]	16	358 198 823	77 360	2765.9729	13	369 298 257	77 908	3590.4924	1.30
LKS	(0, 100]	1	97	2	<0.0001	1	105	2	<0.0001	-
	(100, 1000]	2	217 079	392	0.1513	2	248 085	392	0.1338	0.88
	(1000, 5000]	5	919 439	2138	0.8720	5	1 090 661	2138	0.8571	0.98
	(5000, 10000]	4	14 688 822	6968	52.1697	4	16 871 135	6968	65.2747	1.25
	(10000, 50000]	9	84 805 293	21 354	657.5254	9	96 315 339	21 354	813.3246	1.24
	(50000, 114046.0]	4	303 224 888	77 273	3311.8612	4	342 731 967	77 273	4269.5516	1.29
W	(1000, 5000]	3	1 559 883	2736	1.1861	3	1 756 673	2736	1.1658	0.98
	(5000, 10000]	1	17 269 682	7207	25.2965	1	19 935 110	7207	28.9953	1.15
	(10000, 50000]	5	47 526 608	19 201	302.5079	5	54 511 221	19 201	376.7731	1.25
	(50000, 127115.0]	4	434 112 515	81 456	5442.6880	2	393 634 551	72 520	6681.8820	1.23
E	(100, 1000]	5	127 152	409	0.0847	5	139 832	409	0.0692	0.82
	(1000, 5000]	7	2 348 846	2591	3.1192	7	2 737 457	2591	3.0405	0.97
	(5000, 10000]	4	7 835 916	6859	17.8427	4	9 075 400	6859	20.5916	1.15
	(10000, 50000]	14	87 772 707	28 490	548.3961	13	94 471 567	27 677	668.5759	1.22
	(50000, 143480.0]	6	321 977 858	88 945	2406.2628	5	309 960 716	80 833	3039.7481	1.26
CTR	(1000, 5000]	1	14 486 282	4146	58.6911	1	15 411 167	4146	68.1618	1.16
	(5000, 10000]	2	20 372 008	8373	89.8933	2	23 043 187	8373	111.4469	1.24
	(10000, 50000]	3	94 480 721	24 429	548.3601	3	108 506 395	24 429	667.3145	1.22
	(50000, 136768.0]	2	192 533 755	92 995	1603.8631	2	220 476 147	92 995	2044.6995	1.27

9.9 CONCLUSION

This chapter introduces the *Targeted Multiobjective Dijkstra Algorithm* (T-MDA) a variant of the MDA that uses A* techniques to guide the search towards the target. Moreover, its most notable novelty is the pseudo-lazy management of explored paths. Explored paths that are not in the queue and cannot be discarded are stored in so called $\mathcal{NQP}$ lists. There is one $\mathcal{NQP}$ list for every arc in the graph and the $\mathcal{NQP}$ list for arc a stores the explored paths that are not in the queue and whose last arc is a. This arc based indexing allows the T-MDA to keep the size of its priority queue bounded and to exploit the lexicographic sorting of paths therein to maintain the $\mathcal{NQP}$ lists also sorted using only constant time prepend and append operations.

The instances and the settings used in the computational experiments in this section mirror those conducted in Section 6.3.1. We compare the T-MDA with an enhanced version of the NAMOA$^*_{\text{dr}}$ algorithm (Pulido et al., 2015). The new version is called NAMOA$^*_{\text{dr}}$-lazy algorithm because it maintains explored paths in the algorithm's queue in a lazy way. This queue management was recently shown to be very efficient in the biobjective scenario and indeed our new NAMOA$^*_{\text{dr}}$-lazy algorithm clearly outperforms the NAMOA$^*_{\text{dr}}$.

Our results show that the NAMOA$^*_{\text{dr}}$-lazy is outperformed by the new T-MDA algorithm consistently. The T-MDA algorithm is also able to solve more hard instances than the NAMOA$^*_{\text{dr}}$-lazy algorithm. Its featured combination of a polynomially bounded queue size while still keeping other explored paths in lex. ordered lists turns out to work in practice.

Our settings allow direct comparison of the MDA with the T-MDA. In fact, the modifications are worth it since on road networks, the biggest considered networks, we manage to solve more instances than before in the same amount of time. Those instances solved already by the MDA are solved faster by the T-MDA.

The running time improvements achieved in this chapter make the One-to-One MOSP model a more attractive choice. Given that we are not conducting any type of preprocessing other than the computation of a heuristic, we can expect the running times to improve notably if we manage to transfer speedup techniques from the One-to-One Shortest Path literature to the multiobjective scenario.

Finally, note that the implementations in (Maristany de las Casas, 2023b) can be compiled to solve instances with any number arc cost functions. The folders multidimensional/4obj_results and multidimensional/5obj_results contain results collected on NetMaker and Grid instances using the T-MDA and the NAMOA$^*_{\text{dr}}$-lazy algorithms. The results mirror those presented in this chapter for the three-dimensional case with slightly better speedups in favor of the T-MDA. An in-depth study of higher-dimensional One-to-One MOSP instances like in the publication by (Paixão & Santos, 2013) is still open.

10 DYNAMIC MULTIOBJECTIVE SHORTEST PATH PROBLEMS

Once we have discussed how to tune the MDA to obtain best possible performance in practice in Chapter 9, the next step in our transition from theory to (transportation) applications is to discuss MOSP instances with *dynamic arc cost functions*. The setting in this chapter generalizes the *Time-Dependent Shortest Path* (TDSP) problem (see e.g., Delling & Wagner, 2009; Nannicini, 2009; Foschini et al., 2012) as follows.

In the single-criterion Shortest Path problem, and in the MOSP problem as described so far, arc cost functions map the arcs in the input graph to a scalar. A d-dimensional MOSP instance considers d such arc cost functions. In the TDSP the *arc cost function* takes a second argument $\tau \in \mathbb{R}_{\geqslant 0}$ that encodes the *state* at which the arc's tail node is reached. Then, the arc cost function outputs the cost for traversing the given arc starting with state τ. As the TDSP name indicates, τ is commonly regarded as the time point at which the considered arc a is about to be traversed. Then, $c(a, \tau)$, the cost of a, is the duration for traversing a starting at time point τ and $\tau + c(a, \tau)$ is the time point at which the traversal ends. At the same time, $\tau + c(a, \tau)$ is the input state for the traversal of an outgoing arc of a's head node. In the multi-objective generalization of the TDSP problem that we discuss in this chapter, the *Dynamic MOSP* (Dyn-MOSP) problem, we deal with d state-dependent arc cost functions which we call *dynamic arc cost functions*. In a transportation application the functions may encode the time, the weight, the tank capacity, the battery charge, the collected price zones, etc. of a vehicle traveling through the input graph and that is about to traverse the considered arc in the graph.

We model the problem for One-to-All and One-to-One scenarios and investigate for what kind of dynamic arc cost functions the MDA can be applied to solve the corresponding Dyn-MOSP instances. The chapter ends with a discussion in Section 10.5 of possible extensions and limitations of our considered model with regard to fully realistic transportation settings.

10.1 LITERATURE OVERVIEW

In this section, we stick to the nomenclature in the literature and we present existent results speaking about *time-dependent* scenarios.

We have to differentiate between two time-dependent settings. In the *time-dependent and time-optimal* setting a fastest path is searched, i.e., we optimize w.r.t. time. In the *time-dependent and cost-optimal* setting time evolves and influences the time-dependent arc cost functions w.r.t. which a cost-optimal path is determined. In other words, these two settings consider time as the input state of arc cost function $c : A \times \mathbb{R}_{\geqslant 0} \to \mathbb{R}_{\geqslant 0}$ but $c(a, \tau)$ is interpreted as time in the *time-dependent and time-optimal* setting and as a cost in the *time-*

dependent and cost-optimal setting. Usually, the TDSP problem refers to the time-dependent and time-optimal setting (see e.g., Delling & Wagner, 2009; Nannicini, 2009; Foschini et al., 2012).

Our generalization of the TDSP problem generalizes the time-dependent and time-optimal scenario to the multiobjective case. The literature considering MOSP instances with time-dependent arc cost functions is scarce. We prove that a subpath optimality principle still holds and label-setting MOSP algorithms like the MDA can be used to solve the problem without modifications other than how arc cost functions are evaluated. A similar setting is used by Disser et al. (2008). He mentions the necessity to tackle this kind of problems on train networks and uses Martins's algorithm to solve them without giving any details on required assumptions, correctness, or complexity bounds.

As in the single-criterion case, the multiobjective generalization of the time-dependent and cost-optimal scenario leads to MOSP problems in which label-setting algorithms do not work because subpath optimality cannot be proven. Kostreva and Lancaster (2002) presented an algorithm for non-monotonically increasing arc cost functions that does not reduce to Dynamic Programming. Hamacher et al. (2006) conduct an in depth study of this setting in the biobjective case. We discuss some ideas on this topic in Section 10.5.

We could not find more relevant references in the literature. A similar problem considers digraphs in which nodes and arcs can be added or deleted during the search for efficient s-t-paths (e.g., da Silva et al. (2023)). This setting however is not considered in this chapter.

10.2 PROBLEM DESCRIPTION

In a time-dependent setting, *going along* an arc $(v, w) \in A$ in a digraph $G = (V, A)$ describes a flow unit moving from v to w along (v, w) with constant speed. We say that the flow *traverses* the arc (v, w). Accordingly, we speak about the traversal of an s-t-path p in G between nodes s, $t \in V$ and mean that a flow unit traverses p's arcs in their order of appearance along p. In our setting the traversal of a path cannot be stopped. Additionally, the costs of an arc corresponds to the evaluation of the arc cost functions at the arc's tail node w.r.t. the state with which the arc's traversal starts. This setting is known, e.g., in (Orda & Rom, 1990), as the *frozen arc model with forbidden waiting*. In this chapter, we call time-dependent arc cost functions *dynamic arc cost functions*. As explained already in the introduction, by doing so, we hint that the components of the states $\tau \in \mathbb{R}^d_{\geqslant 0}$ and $c(a, \tau) \in \mathbb{R}^d_{\geqslant 0}$ can encode as time points (Nannicini, 2009), weights (Blanco et al., 2016; Blanco et al., 2022), battery charges (Baum et al., 2020), already paid tolls (Euler et al., 2022), etc.

Definition 10.1 (Dynamic Costs (cf. Nannicini, 2009, Section 1.2))**.** Let $G = (V, A)$ be a digraph.

DYNAMIC ARC COST FUNCTION A *dynamic arc cost function* is a function

$$\begin{aligned} c : (A \times \mathbb{R}_{\geqslant 0}) &\to \mathbb{R}_{\geqslant 0} \\ (a, \tau) &\mapsto c(a, \tau). \end{aligned} \tag{16}$$

An input pair (a, τ) consists of an arc $a = (u, v)$ and a state $\tau \in \mathbb{R}_{\geqslant 0}$ that represents the state of the flow unit at u. Then, $c(a, \tau)$ represents the costs for traversing a starting at state τ.

DYNAMIC PATH COSTS Let $s, t \in V$ be two nodes, $\tau_0 \in \mathbb{R}_{\geqslant 0}$ an initial state, and c a dynamic arc cost function. Consider an s-t-path p with $k \in \mathbb{N}$ arcs, i.e., $p = (a_1, \dots, a_k)$. For $i \in \{1, \dots, k\}$, set $p_i := (a_1, \dots, a_i)$ to be the prefix path of p until its i^{th} arc. For every i, the *dynamic path cost* of p_i w.r.t. c is defined as

$$c(p_i, \tau_0) := \begin{cases} \tau_0 + c(a_1, \tau_0), \text{ if } i = 1, \\ c(p_{i-1}, \tau_0) + c\big(a_i, c(p_{i-1}, \tau_0)\big), \text{ otherwise.} \end{cases} \tag{17}$$

Finally, the *dynamic cost* of p starting with state τ_0 at s are $c(p, \tau_0) = c(p_k, \tau_0)$.

MULTIOBJECTIVE GENERALIZATION Consider $d \in \mathbb{N}$ dynamic arc cost functions c_i, $i \in \{1, \dots, d\}$ and fix an arc $a \in A$. Let $\tau \in \mathbb{R}^d_{\geqslant 0}$ be a given vector of states. We assume that for every $j \in \{1, \dots, d\}$, the j^{th} entry of τ is the input of the j^{th} arc cost function. I.e., the vector

$$\big(\tau_1 + c_1(a, \tau_1), \dots, \tau_d + c_d(a, \tau_d)\big) \in \mathbb{R}^d_{\geqslant 0}$$

denotes the costs after a's traversal if it is started with state τ. Then if we have d dynamic arc cost functions for every arc in G and a d-dimensional initial vector of states $\tau_0 \in \mathbb{R}^d_{\geqslant 0}$, it is straightforward to generalize (17) to obtain d-dimensional dynamic path costs of p assuming that its traversal starts with state τ_0.

Remark 10.1. ***INPUT OF DYNAMIC ARC COST FUNCTIONS*** *We have chosen to define dynamic arc cost functions depending on two input values: an arc and the state at which we assume the traversal of the input arc begins. This choice is motivated by the fact that in the static arc costs setting in previous chapters, we denoted arc costs by* $c(a) \in R^d$. *In this chapter and also in Chapter 11 we are however interested in properties of the scalar functions* $c(a, \cdot) : \mathbb{R}_{\geqslant 0} \to \mathbb{R}_{\geqslant 0}$, *where a state* τ *is mapped to* $c(a, \tau)$ *and* $a \in A$ *is fixed. We call these functions dynamic arc cost functions too.*

τ_0 ***SUMMAND IN THE RECURSIVE BASE CASE*** *For* $i = 1$ *in (17) we have an affine term* τ_0. *The reason is that we are not interested in measuring e.g., the time that passes since the traversal of a path starts like in an earliest arrival setting often considered in the literature. In our setting we consider time, weight, tank capacity, and so on as values that describe the state of a vehicle traveling through a network. Thus, weather prognoses, consumption functions or similar are evaluated depending not on duration or consumption but on absolute time or weight of the vehicle.*

Often in the literature dynamic arc cost functions are assumed to be periodic to model time-dependent arc costs capturing for example rush hours or timetables. It implies $c(a, \tau + T) = c(a, \tau)$ for any $\tau \in [0, T]$. In other applications like Flight Planning or Electric Vehicle Routing, the dynamic arc cost functions are limited by the tank/battery capacity. In this thesis, we assume that the dynamic arc cost functions are defined on a sufficiently large interval. This simplifies notation and proofs while still capturing all relevant theoretical insights.

Note that if the arc cost functions are constant and if we assume $\tau_0 = 0$, the recursive equation (17) coincides with (4) in Definition 3.1, i.e., $c(p, \tau_0) = c(p)$ for all $\tau_0 \in \mathbb{R}_{\geqslant 0}$. The multiobjective generalization of the TDSP considers multiple dynamic arc cost functions. First, we define our notation to refer to sets of efficient paths.

Definition 10.2 (Notation for Efficient Paths)**.** Let $G = (V, A)$ be a digraph with $d \in \mathbb{N}$ dynamic arc cost functions associated with every arc and let $\tau_0 \in \mathbb{R}^d_{\geqslant 0}$ be an initial state. For two nodes $s, v \in V$ and a set P of s-v-paths, we define $c(P, \tau_0) := \{c(p, \tau_0) \mid p \in P\}$ to be the set of dynamic path costs induced by the paths in P. Moreover, we use the notation $P^*(\tau_0) \subseteq P$ to denote a minimal complete set of efficient s-v-paths in P w.r.t. the d-dimensional dynamic path costs obtained depending on τ_0. Given that we denote the set of all s-v-paths in G by P_{sv}, we have that $P^*_{sv}(\tau_0)$ is a minimal complete set of s-v-paths in G w.r.t. the initial state τ_0.

Finally, we define the dynamic costs generalization of MOSP as follows.

Definition 10.3 (Dynamic MOSP problems)**.** Consider a digraph $G = (V, A)$, a source node $s \in V$, d dynamic arc cost functions c_i, $i \in \{1, \dots, d\}$, and a d-dimensional vector $\tau_0 \in \mathbb{R}^d_{\geqslant 0}$ of initial states.

ONE-TO-ALL DYNAMIC MOSP The *One-To-All Dynamic MOSP* problem is to find a minimal complete set $P^*_{sv}(\tau_0)$ of efficient s-v-paths for every $v \in V$ w.r.t. the paths' cost vector induced by the functions c_i, $i \in \{1, \dots d\}$. Thereby, every path starts at s with state τ_0. We call a tuple $\mathcal{I} = (D, s, d, [c_i]_{i=1}^d, \tau_0)$ a One-to-All Dynamic MOSP instance.

ONE-TO-ONE DYNAMIC MOSP If additionally a target node $t \in V$ is specified in the input, the *One-to-One Dynamic MOSP* problem is to find a minimal complete set $P^*_{st}(\tau_0)$ of s-t-paths w.r.t. c_i as in the One-to-All case. We call a tuple $\mathcal{I} = (D, s, t, d, [c_i]_{i=1}^d, \tau_0)$ a One-to-One Dynamic MOSP instance.

Whenever we use a Dynamic MOSP (Dyn-MOSP) instance without specifying whether we consider the One-to-All or the One-to-One case, the statement is valid in both scenarios. Moreover, in the remainder of this chapter, for an arc a and a state $\tau \in \mathbb{R}^d$, we write $c(a, \tau) := (c_1(a, \tau_1), \dots, c_d(a, \tau_d))$. Hence, we assume that the i^{th} dynamic arc cost function $c_i(a, \cdot)$ of a depends on the i^{th} coordinate of τ only. Accordingly, we write $c(p, \tau) = (c_1(p, \tau), \dots c_d(p, \tau))$ for the cost vector of a path p.

Example 10.1. Figure 22 shows an example of a biobjective Dyn-MOSP instance and how the costs of two paths evolve during their traversal. We

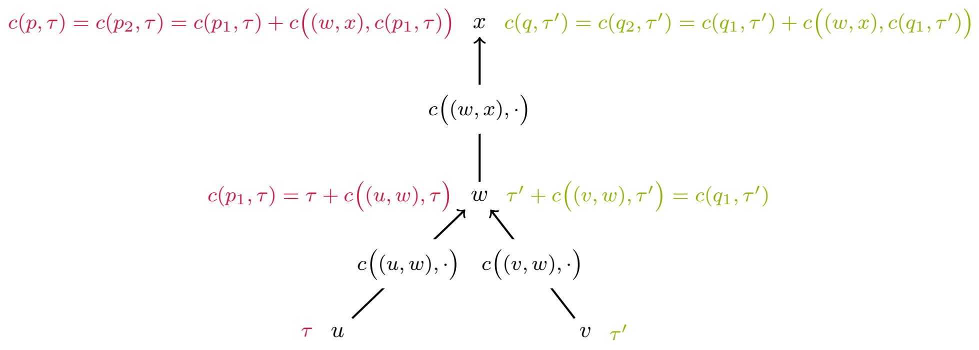

$c(p,\tau) = c(p_2,\tau) = c(p_1,\tau) + c\big((w,x), c(p_1,\tau)\big)$
x
$c(q,\tau') = c(q_2,\tau') = c(q_1,\tau') + c\big((w,x), c(q_1,\tau')\big)$
$c\big((w,x),\cdot\big)$
$c(p_1,\tau) = \tau + c\big((u,w),\tau\big)$
w
$\tau' + c\big((v,w),\tau'\big) = c(q_1,\tau')$
$c\big((u,w),\cdot\big)$
$c\big((v,w),\cdot\big)$
τ
u
v
τ'

Figure 22: Graph corresponding to the Dyn-MOSP instance from Example 10.1.

consider the u-x-path $p = ((u,w),(w,x))$ with initial state τ at u and the v-x-path $q = ((v,w),(w,x))$ with initial state τ' at v. The costs of p and q are depicted in red and green, respectively.

10.3 LABEL SETTING DYN-MOSP ALGORITHM

Label setting MOSP algorithms like the MDA can be applied for Dyn-MOSP instances if the dynamic arc cost function fulfill the Bellman condition from Theorem 3.1. To this aim, we impose the *First-In-First-Out (FIFO)* condition on every dynamic arc cost function.

Definition 10.4 (First-In-First-Out (FIFO) Condition (e.g. Nannicini, 2009, Section 1.2.2)). Consider a digraph $G = (V,A)$ and for a fixed arc $a \in A$, a dynamic arc cost function $c(a,\cdot) : \mathbb{R}_{\geqslant 0} \to \mathbb{R}_{\geqslant 0}$. Then, c fulfills the *first-in-first-out condition* and is called a *FIFO function* if for $\tau,\ \tau' \in \mathbb{R}_{\geqslant 0}$

$$\tau < \tau' \Rightarrow \tau + c(a,\tau) \leqslant \tau' + c(a,\tau'). \tag{18}$$

As noted in e.g., Nannicini (2009, Section 1.2.2), the FIFO condition can be easily checked for differentiable arc cost functions.

Lemma 10.1. *Let $a \in A$ be fixed. A differentiable arc cost function $c(a,\cdot)$ is a FIFO function if the derivative $c'(a,\cdot)$ fulfills $c_i'(a,\cdot) \geqslant -1$.*

Besides the easy check to determine whether a given arc cost function is a FIFO function, the last result has desirable implications for the applicability of Dyn-MOSP problems in practice. The fact that FIFO functions do not need to be monotonically non-decreasing widens the range of applications. For example if battery consumption is an optimization criterion, the battery recharging capability of electric vehicles while breaking reduces the cost during paths' traversal.

The following theorem implies that with FIFO functions in its input, Dyn-MOSP instances can be solved using the label-setting MOSP algorithms discussed so far in this thesis.

Theorem 10.1. *Consider a Dyn-MOSP instance $\mathcal{I}$ with d FIFO functions c_i. For every node $v \in V$ there exists a minimal complete set $P^*_{sv}(\tau_0)$ of efficient s-v-paths s.t. for every $p \in P^*_{sv}(\tau_0)$ all s-u-subpaths of p, $u \in p$, are efficient s-u-paths.*

Proof. Consider a minimal complete set $P^*_{sv}(\tau_0)$ of efficient s-v-paths for some $v \in V$ and let $p \in P^*_{sv}(\tau_0)$. Assume that $u \in p$ is the first node along p for which the s-u-subpath $p^{s\to u}$ of p is not an efficient path. Then, there exists an s-u-path q that dominates $p^{s\to u}$. We have: $c_i(q, \tau_0) \leqslant c_i(p^{s\to v}, \tau_0)$ for all $i \in \{1, \dots, d\}$ and $c_j(q, \tau_0) < c_j(p^{s\to v}, \tau_0)$ for at least one $j \in \{1, \dots, d\}$. Assume that $a = (u, w) \in A$ is the outgoing arc of u in p. Then, the FIFO property implies for all $i \in \{1, \dots, d\}$

$$\begin{aligned} c_i(q \circ a, \tau) &= c_i(q, \tau) + c_i\big(a, c_i(q, \tau)\big) \\ &\leqslant c_i(p^{s\to u}, \tau) + c_i\big(a, c_i(p^{s\to u}, \tau)\big) = c_i\big(p^{s\to u} \circ a, \tau\big). \end{aligned} \tag{19}$$

If the inequality is not strict for any coordinate, we have $c_i(q \circ a, \tau) = c_i(p^{s\to u} \circ a, \tau)$. In this case, we can substitute p in $P^*_{sv}(\tau)$ by the s-v-path $q \circ a \circ p^{w\to v}$. If q contains any subpath starting at s that is not efficient, we repeat the arguments from this proof to find a suitable minimal complete set $P^*_{su}(\tau)$ of efficient s-u-paths. If all subpaths of q are efficient, we have proven the statement.

Otherwise if there is an index $j \in \{1, \dots, d\}$ for which the inequality (19) is strict, we can continue expanding $p^{s\to w} = p^{s\to u} \circ a$ and $q \circ a$ along $p^{w\to v}$. Due to the FIFO conditions, there must be an arc along $p^{w\to v}$ after which both paths become cost-equivalent and we repeat the argument from the last paragraph. If that is not the case, $c_j(q \circ p^{w\to v}, \tau) < c_j(p, \tau)$ which contradicts the assumption of p being an efficient s-v-path. □

The statement implies that for a Dyn-MOSP algorithm it is sufficient to store only efficient paths since from them, efficient paths for successor nodes can be built. In other words, dominated subpaths can be discarded and label-setting algorithms like the MDA can be applied. To prove the correctness of the MDA in Section 4.2, we needed the statements in Lemma 4.1 to Lemma 4.5. These facts prove that the algorithm makes paths permanent in nondecreasing order w.r.t. the total order $\prec_Q$ and that it is label-setting. For the proofs we only use the non-negativity of the cost vectors and the compatibility (cf. Definition 3.5) of the used total order $\prec_Q$ and the dominance or equivalence order $\preceq_D$. Hence, these statements are also valid in a Dyn-MOSP setting.

Theorem 4.1 and Theorem 4.2 are the statements that actually prove the MDA's correctness. Besides the aforementioned lemmas, the proves only require the Bellman condition for subpath efficiency from Theorem 3.1. Hence, given Theorem 10.1 in the dynamic costs setting, the MDA or the T-MDA are Dyn-MOSP algorithms if we modify the evaluation of the arc cost functions according to Definition 10.1.

Theorem 10.2. *Consider a One-to-All or a One-to-One Dyn-MOSP instance $\mathcal{I}$ with d FIFO functions. The MDA or the T-MDA, respectively, solve $\mathcal{I}$.*

Remark 10.2 (Heuristics for One-to-One Dyn-MOSP). *In Section 9.3 we discussed how to compute heuristics to speed up One-to-One MOSP queries. In order to compute the paths' reduced costs and guarantee the ordering of the paths w.r.t. $\preceq_D$, the i^{th} heuristic value $\pi_i(v)$ for a node $v \in V$ needs to be an underestimation of the costs of an v-t-path in the i^{th} cost dimension. The discussed heuristic was* *best possible in the sense that, since we computed the shortest v-t-distance in every dimension, no tighter underestimator exists.*

The situation is different in a dynamic costs scenario. Whether an arc $a = (u, v) \in A$ is traversed with costs $\min_{\tau \in \mathbb{R}_{geq}} c_i(a, \tau)$ depends on the state τ. Thus, the heuristic built using this minimum arc cost for every arc is often bad. Another issue is that in Section 9.3 we computed heuristics using a backward search from t. However, in a dynamic costs scenario the state at which the backward search needs to be started is unknown. This problem also arises in the single-criterion time-dependent Shortest Path problem (cf. Delling & Wagner, 2009).

In general, finding good heuristics in time-dependent/dynamic costs scenarios is thus an application dependent task as shown for example in (Blanco et al., 2022).

10.4 COMPLEXITY

Computing a path's costs recursively as in formula (17) for a path p with ℓ arcs requires $\mathcal{O}(d\ell)$ time. However, label-setting algorithms like the MDA build paths incrementally arc by arc. Thus, if (u, v) is the last arc of p, the costs $c(p^{s \to u}, \tau) \in \mathbb{R}^d_{\geqslant 0}$ is known and

$$c(p, \tau) = c(p^{s \to u}, \tau) + c((u, v), c(p^{s \to u}, \tau))$$

requires only the evaluation of (u, v)'s d arc cost functions.

In the asymptotic running time bound of Dyn-MOSP algorithms, the evaluation of dynamic arc cost functions cannot be neglected in general. If a dynamic arc cost function c is given as an analytic formula, we assume that $c(a, \tau)$ is evaluated in $\mathcal{O}(1)$ time and also its storage requires $\mathcal{O}(1)$ space.

Most often in the literature (cf. Delling et al., 2009; Blanco et al., 2016), c is given as a d piecewise functions, each of them represented in a lookup table. The entries in the tables are the functions' breakpoints and an interpolation rule between these breakpoints is given as an analytic formula. In this scenario, we assume that the number of breakpoints defining c is in $\mathcal{O}(2^n)$. Moreover, we assume that the relevant breakpoints to evaluate c at any point $\tau \in \mathbb{R}_{\geqslant 0}$ can be found using binary search. If that is the case, $c(a, \tau)$ can be evaluated in $\mathcal{O}(d \log 2^n) = \mathcal{O}(dn)$ for any $a \in A$. Then, for any path s-v-path p, calculating the costs of an expansion q of p along an outgoing arc $(v, w) \in \delta^+(v)$ in the MDA is a $\mathcal{O}(dn)$ operation. Since we store paths as labels (cf. Section 3.3)in the output sets $P_{sw}(\tau_0)$ of the MDA, the costs of q are the only costs that we need to calculate in the dominance check $c(P_{sw}(\tau_0)) \preceq_D c(q, \tau_0)$ in Line 3 of propagate and in Line 7 of nextQueuePath. Since dominance checks run in $\mathcal{O}(N_{\max})$ and in general $N_{\max}$ is exponential in the input size, the $\mathcal{O}(dn)$ operations to calculate $c(q)$ vanish in the asymptotic running time bound of propagate and nextQueuePath.

All in all, we state the following assumption for the remainder of this chapter and also for Chapter 11.

Assumption 10.1. We consider Dyn-MOSP instances with piecewise linear and FIFO arc cost functions. Each function is assumed to have at most n breakpoints.

Using this assumption, the time complexity of the MDA for the One-to-One and the One-to-All Dyn-MOSP problem remains as in Theorem 4.4.

Corollary 10.1. *The MDA and the T-MDA for Dyn-MOSP instances run in*

$$\begin{aligned} &\mathcal{O}\left(N \log dn + N_{\max}^2 dm\right), \quad \textit{for } d > 2 \textit{ and} \\ &\mathcal{O}\left(N \log dn + N_{\max} dm\right), \quad \textit{for } d = 2 \end{aligned} \tag{20}$$

Similarly, the space consumption can be derived as in Theorem 4.6 but we need to consider the space for storing the are cost functions as lookup tables. In contrast to the asymptotic running time bound in which we can neglect the evaluation arc cost functions because we *hide* it in a logarithmic term, the storage of the tables is asymptotically relevant.

Corollary 10.2. *Consider a Dyn-MOSP instance $\mathcal{I}$ such that for $i \in \{1, \dots, d\}$, the maximum number of breakpoints in a dynamic arc cost function is $B_i < n$. Then, storing all dynamic arc cost functions requires $\mathcal{O}\left(m \sum_{i=1}^{d} B_i\right)$ and the memory consumption of the MDA while solving $\mathcal{I}$ is in*

$$\mathcal{O}\left(N + dmn\right). \tag{21}$$

10.5 DYN-MOSP FOR TRANSPORTATION APPLICATIONS

In this section we give a more general view into Dyn-MOSP variants that are relevant for applications. As mentioned in this chapter's introduction, there are also time-dependent Shortest Path problems in which arc cost functions are time-dependent but time is not optimized. We called this setting *time-dependent and cost-optimal*. Its single-criterion version defines two time-dependent functions on the arcs. An arc's first function captures how time evolves when traversing the arc. The arc's second function depends only on time but it returns the cost of traversing the arc starting at the given time.

Definition 10.5 (Time-Dependent and Cost-Minimal Shortest Path Problem (Foschini et al., 2012; Hamacher et al., 2006; Orda & Rom, 1990)). Consider a digraph $G = (V, A)$, a source node $s \in V$, a starting time point $\tau_0 \in \mathbb{R}_{\geqslant 0}$, and two functions time, cost for every arc as in Assumption 10.1. Given an s-u-path p for some $u \in V$ reaching u at $\mathrm{time}(p, \tau_0)$ (cf. Definition 10.1) with cost $\mathrm{cost}(p, \tau_0)$ and an arc $(u, v) \in \delta^+(u)$, the time-dependent cost of $p \circ (u, v)$ is

$$\mathrm{cost}\big(p \circ (u,v), \tau_0\big) := \mathrm{cost}(p, \tau_0) + \mathrm{cost}\big((u,v), \mathrm{time}(p, \tau_0)\big).$$

Time evolves as in Definition 10.1 and thus:

$$\mathrm{time}\big(p \circ (u,v), \tau_0\big) := \mathrm{time}(p, \tau_0) + \mathrm{time}\big((u,v), \mathrm{time}(p, \tau_0)\big).$$

The One-to-All *time-dependent and Cost Minimal Shortest Path* (TD-CM-SP) problem is to find a cost-optimal s-v-path for every $v \in V$.

Example 10.2 and Figure 23 contain an example of a TD-CM-SP instance taken from (Hamacher et al., 2006). The example is used in the original publication to show how solutions in the TD-CM-SP problem do not adhere to a subpath optimality principle.

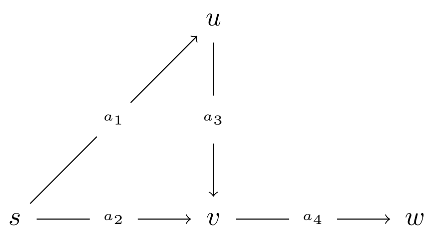


Figure 23: Graph corresponding to the TD-CM-SP instance used in Example 10.2. The arc time and cost functions are in Table 12.

Arc	time	cost
a_1	3	1
a_2	1	5
a_3	1	1
a_4	1	$2\tau + 2$

Table 12: time and cost functions of the arcs in the graph in Figure 23. A state (τ, γ) at an arc's tail is mapped to $(\tau + \mathrm{time}(\tau), \gamma + \mathrm{cost}(\tau))$.

Example 10.2 (Broken subpath optimality (cf. Hamacher et al., 2006))**.** In the TD-CM-SP instance shown in Figure 23 and in Table 12, all functions are FIFO. The One-to-One time-dependent Shortest Path problem in a time-dependent and cost-optimal setting seeks to find an s-w-path with minimal cost starting at s at time 0 with cost 0. The path $p = ((s,v),(v,w))$ is the only optimal solution in this context. We have $\mathrm{time}((s,v),0) = 1$, $\mathrm{cost}((s,v),0) = 5$, and

$$\mathrm{time}(p,0) = \mathrm{time}\big(p^{s\to v} \circ (v,w),0\big) = 1 + \mathrm{time}\big((v,w),1\big) = 1+1 = 2$$
$$\mathrm{cost}(p,0) = \mathrm{cost}\big(p^{s\to v} \circ (v,w),0\big) = 1 + \mathrm{cost}\big((v,w),1\big) = 5+4 = 9.$$

However, for the s-v-path $q = ((s,u),(u,v))$ we have $\mathrm{time}(q,\tau_0) = 4$ and $\mathrm{cost}(q,\tau_0) = 2$. Hence, the s-u-subpath $p^{s\to u}$ of p is not a cost-optimal s-u-path and the cost-optimal s-w-path p is not built out of optimal subpaths.

The problem in Example 10.2 is easy to see: even though optimality is decided based on one criterion only, this criterion depends on the time function. In the example time and cost rather uncorrelated and thus, reaching node v via u is worse in time and better in cost. However, the late arrival penalizes the cost increase along the arc (v,w) causing the path p from the example to be the only optimal one.

An immediate solution to this problem is to model the TD-CM-SP problem as a biobjective problem. Since the cost functions are time-dependent but FIFO, there holds: $\tau < \tau' \Rightarrow \mathrm{cost}(a,\tau) \leqslant \mathrm{cost}(a,\tau')$ for time values $\tau, \tau' \in \mathbb{R}_{\geqslant 0}$ and every $a \in A$. With this and following the arguments used in Section 10.3, it is straightforward to conclude that the TD-CM-SP problem can be solved using a biobjective version of the MDA for Dyn-MOSP problems. The asymptotic time and space bounds coincide with (20) and (21), respectively. These are good and bad news.

The bad news are that the fastest path problem (i.e., time-dependent and time-optimal) with forbidden waiting and using the frozen arc model is solvable using variants of label-setting Shortest Path algorithms (cf. Delling & Wagner, 2009) and thus, polynomially solvable. The TD-CM-SP problem on the other hand using a biobjective version of the MDA for Dyn-MOSP problems is solved using an output sensitive algorithm.

Despite this increased asymptotic complexity bound, the biobjective MDA is very efficient in practice and the $\mathcal{NP}$-hardness of the TD-CM-SP problem has already been proven by Orda and Rom (1990).

A multiobjective generalization of the TD-CM-SP problem in which d cost functions are given and all of them are time-dependent is solvable as described above using the $(d+1)$-dimensional version of the MDA for Dyn-MOSP instances. We only require that the arc cost function adhere to Assumption 10.1. As in the TD-CM-SP problem, the $(d+1)^{\text{th}}$ objective in this scenario is time.

10.5.1 Higher-dimensional function inputs

Other variants of Dyn-MOSP problems cannot be solved using label-setting algorithms so easily. In realistic Flight Planning, Electric Vehicle Routing, Public Transportation, etc. cost functions are rarely dependent on time only. E.g., the fuel or battery consumption depends on time because of weather and traffic but it also depends on the fuel load and/or the battery charge.

The applicability of label-setting algorithms or other Dynamic Programming approaches in these scenarios is not straightforward mainly because it is unclear how to generalize the notion of FIFO functions to dynamic arc cost functions in which the i^{th}, $i \in \{1, \dots, d\}$, cost function depends on a subset of the input state components.

Whether a FIFO generalization works will depend, among other things, on how correlated the considered states and functions are, how the partial derivatives of the cost functions relate to each other, etc. Problem dependent modeling is required. In fact, to the best of our knowledge this highly relevant problem for the applicability of MOSP in transportation has not been studied in the literature. The propagation of state of charge functions (Baum et al., 2020) or arrival time functions involving piecewise function minimization (Delling & Wagner, 2009) comes the closest to the mentioned setting but considers only one optimization criterion.

10.6 EXPERIMENTS

We could not find openly available data for Dyn-MOSP instances. After the next chapter about approximation algorithms for Dyn-MOSP problems, we report our results obtained from *Horizontal Flight Planning* instances using an approximation algorithm for Dyn-MOSP based on the MDA. Since we have proven that Dyn-MOSP instances with FIFO functions can be solved using the MDA in a straightforward way, we do not need large scale experiments focusing on the performance of the MDA to solve Dyn-MOSP instances.

11 MULTIOBJECTIVE DIJKSTRA FPTAS

This chapter is based on the publication (Maristany de las Casas, Borndörfer, et al., 2021). Background on *approximation algorithms* can be read in Williamson and Shmoys (2009). Whenever we consider a Dyn-MOSP instance in this chapter without specifying whether it is a One-to-One or a One-to-All instance, the corresponding statement holds for both types of problems. Moreover, we assume that dynamic arc cost function fulfill Assumption 10.1.

Recall that the principal intractability in Dyn-MOSP problems arises from the possibly exponential number of efficient paths (w.r.t. the instances' input size). That is why it is natural to be interested in approximation algorithms that compute a *meaningful* set of solutions that is *good enough* and has *polynomially bounded* cardinality. An intuitive approach is to subdivide the outcome space into a polynomial number of hyperrectangle and only allow one solution per cube in the algorithms' output. However, the sizes of the cubes have to be carefully chosen to bound the error that arises when comparing minimal complete sets of efficient paths and the paths output by an approximation algorithm.

Example 11.1. Consider a MOSP instance with constant bidimensional arc costs. For every node $v \in V$, we partition the outcome space $c(P_{sv}) \subset \mathbb{R}^d_{\geqslant 0}$ into 2×2 squares as the dotted lines show in Figure 24. Moreover, assume that we design a variant of a MOSP algorithm that keeps only the lex. smallest s-v-path in every cell.

Assume a node u can be reached only via two paths p_1 and p_2 with costs $(2.1, 3.9)$ and $(3.9, 2.1)$, respectively. Then, p_1 and p_2 would be stored in the same square and our fictive algorithm keeps only p_1.

The second relevant node in this example is $v \in \delta^+(u)$. The arc $a = (u, v)$ has costs $c(a) = (1, 1)$. Consider the extensions $q_i = p_i \circ (u, v)$, $i \in \{1, 2\}$. Since $c(q_1) = (3.1, 4.9)$ and $c(q_2) = (4.9, 3.1)$, q_1 and q_2 do not share a square in the outcome space partition of $c(P_{sv})$ (see Figure 24).

Besides q_1 and q_2 there is one more s-v-path in the considered graph: the s-u-path q with $c(q) = (M, 3.9)$ for some large $M \in \mathbb{R}_{\geqslant 0}$. The path q_1 does not dominate q but q_2 dominates q. However, since our fictive algorithm discards p_2, q_2 is not considered. Thus, when the path q is built, it is not dominated. The error in the c_1 cost dimension for discarding q_1 is in $\mathcal{O}(M)$.

We combine an outcome space partition technique from the literature with the MDA to design a new *Fully Polynomial Time Approximation Scheme* (FPTAS) for Dyn-MOSP problems. Note that this problem class includes MOSP problems with constant arc cost vectors.

The generalization of the definition of an FPTAS to the multiobjective scenario is straightforward and has already been discussed in the literature (e.g., Papadimitriou & Yannakakis, 2000; Tsaggouris & Zaroliagis, 2007). To streamline our exposition, we repeat relevant definitions in Section 11.1. The following assumptions are also common in the literature.

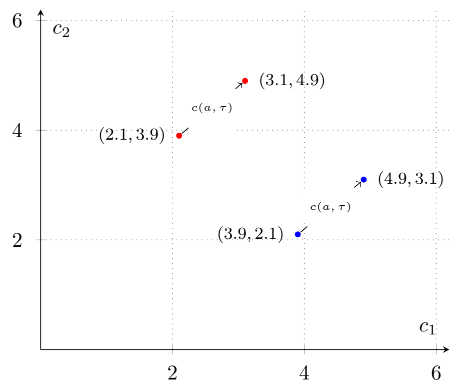


Figure 24: Situation described in Example 11.1. The lower left red and blue dots correspond to the costs of p_1 and p_2. The upper right red and blue dots correspond to the costs of q_1 and q_2. We observe that the lower left dots share a square. After the shift of both cost vectors by the costs of the arc a, the upper right dots are in different squares.

Assumption 11.1. In this chapter, we adhere to following conventions regarding the dimensionality of the considered MOSP instances and the sign of the arc cost functions.

- The running times of the algorithms discussed in this section are exponential in the number d of cost functions. We consider d as a constant.
- We consider Dyn-MOSP instances with positive arc cost functions only.

11.1 NOTATION AND GENERAL DEFINITIONS

Approximate solutions are defined as follows.

Definition 11.1 (Covers (cf. Papadimitriou & Yannakakis, 2000, Section 2))**.** Consider a combinatorial multiobjective optimization problem with d minimization objectives. Let X be the set of feasible solutions and for any $x \in X$, denote its cost vector by $c(x) \in \mathbb{R}^d_{>0}$. Moreover, let $\alpha \in \mathbb{R}_{\geqslant 1}$ be an approximation ratio.

SOLUTION COVER For two feasible solutions $x, y \in X$, x *α-covers* y if

$$c(x) \leqslant \alpha c(y).$$

α-COVER A subset $X_\alpha \subseteq X$ is an *α-cover* of X if for every solution $x \in X$ there exists a solution $\bar{x} \in X_\alpha$ s.t. $\bar{x}$ α-covers x.

The notion of α-covers relaxes the dominance relation between solutions, i.e., paths in our MOSP setting. If a path p dominates a path q, p α-covers q for any feasible value of α. However if the costs of p are *similar* to those of q but p does not dominate q, then p α-covers q if, after worsening q's costs by the (small) factor α, the resulting cost vector is dominated by the costs of p.

Note that in Definition 11.1 we have chosen a scalar approximation factor α. This choice is only to ease the notation in the remainder of the chapter.

The results in this section can easily be extended to d-dimensional vectors of approximation ratios. In practice this is actually needed to approximate e.g., time and weight in a meaningful way. However, it makes also sense to define *the* approximation ratio for the approximation algorithm at hand as the largest allowed error.

Recall that given a Dyn-MOSP instance, we denote the set of s-v-paths in the input digraph G by P_{sv}. We use the notation $P^*_{sv}(\tau_0)$ to refer to a minimal complete set of efficient s-v-paths starting at node s with state τ_0. In this thesis, we have already used covers as explained in the next example.

Example 11.2. Consider a Dyn-MOSP instance $\mathcal{I}$ as in Definition 10.3, fix a node $v \in V$ and let $\alpha \in \mathbb{R}^d_{\geqslant 1}$ be given. Any minimal complete set $P^*_{sv}(\tau_0)$ of efficient s-v-paths is a 1-cover of P_{sv}. Every s-v-path p that is not efficient is dominated by an efficient s-v-path $p^* \in P_{sv}$. Thus, p^* 1-covers p. Moreover if $p^* \notin P^*_{sv}(\tau_0)$, there exists a solution $q^* \in P^*_{sv}(\tau_0)$ s.t. $c(p^*, \tau_0) = c(q^*, \tau_0)$. This implies that q^* 1-covers p^*.

Definition 11.2 (FPTAS for Multiobjective Optimization Problems (cf. Papadimitriou & Yannakakis, 2000, Section 2))**.** Let $\mathcal{P}$ be a d-dimensional multiobjective combinatorial optimization problem for some $d \in \mathbb{N}$ and $\mathcal{I}$ any instance of $\mathcal{P}$. A *Fully Polynomial Time Approximation Scheme* (FPTAS) for $\mathcal{P}$ is a family of algorithms $(\mathcal{A}_\varepsilon)_{\varepsilon \in \mathbb{R}^d_{>0}}$ s.t. for every fixed and constant vector $\varepsilon \in \mathbb{R} > 0$

- $\mathcal{A}_\varepsilon$ returns a $(1+\varepsilon)$-cover of the feasible solutions set X of $\mathcal{I}$ and
- the running time and space consumption A_ε are polynomially bounded by the input size of $\mathcal{I}$ and by $\frac{1}{\varepsilon}$.

In the remainder of this chapter, whenever we refer to a $(1+\varepsilon)$*-cover of the set of feasible solutions* at hand and there is no risk of misunderstanding, we will just write a $(1+\varepsilon)$*-cover*.

Assumption 11.2. As in (Breugem et al., 2017; Papadimitriou & Yannakakis, 2000; Tsaggouris & Zaroliagis, 2007), we assume that $\varepsilon \leqslant 1$ such that $\ln(1+\varepsilon) \in \Theta(\varepsilon)$.

11.2 LITERATURE REVIEW

The contents in this section are partially copied from (Maristany de las Casas, Borndörfer, et al., 2021, Section 1.1.). For BOSP problems ($d = 2$) with constant arc cost functions, Hansen (1980) introduced the first known FPTAS for a MOSP problem variant. It runs in

$$\mathcal{O}\left(\frac{(mn)^2}{\varepsilon}\log\frac{n^2}{\varepsilon}\right). \tag{22}$$

It uses multiple calls to his label-setting BOSP algorithm (a biobjective version of Martins's algorithm (E. Q. V. Martins, 1984)) after scaling the first cost component of the original arc costs by a factor $\frac{\varepsilon}{n}$ (cf Hansen, 1980, Algorithm 4). Warburton (1986) introduced the first FPTAS for MOSP problems on Directed Acyclic Graphs. FPTAS for MOSP problems on general digraphs use partitions of the outcome space to guarantee the polynomial running time of the approximation algorithms.

OUTCOME SPACE PARTITIONS. In the more general field of approximation algorithms for multiobjective combinatorial optimization problems, Papadimitriou and Yannakakis (2000) set a milestone. They proved that for a combinatorial multiobjective optimization problem with d objectives and an $\varepsilon > 0$, a $(1+\varepsilon)$-cover exists. The idea is to subdivide the outcome space into hyperrectangles whose lengths depend on ε and on the input cost vectors. Then, at most one solution per cell is allowed and the subdivision guarantees that a solution in a cell is a $(1+\varepsilon)$ cover of any other solution assigned to the cell. How to choose the subdivision correctly is discussed later in Section 11.3. Having this result, the question of whether this cover can be computed in polynomial time w.r.t. the input size and w.r.t. ε was open. In (Papadimitriou & Yannakakis, 2000, Theorem 2) the authors prove that the answer is affirmative if and only if given a vector $b \in \mathbb{R}^d_{\geqslant 0}$, an algorithm can answer that there is a solution x with $c(x) \leqslant b$ or that there is no solution y with $c(y) \leqslant (1+\varepsilon)b$. For MOSP problems, this is the case.

THE OUTCOME SPACE PARTITION FOR MOSP. The outcome space partition approach for the design of FPTAS for MOSP was later used by Tsaggouris and Zaroliagis (2007). Their algorithm combines the partition approach with the classical Bellman Ford algorithm for Shortest Path problems. This produces a $(1+\varepsilon)$-*cover* of the exact set of efficient paths. We set

$$C := \max_{i \in \{1,\dots,d\}} \left\{ \frac{\max_{a \in A}\{c_i(a)\}}{\min_{a \in A}\{c_i(a)\}} \right\}.$$

As stated in (Tsaggouris & Zaroliagis, 2007, Table 1), for $d = 2$ their algorithm runs in

$$\mathcal{O}\left(\frac{n^2 m \log(nC)}{\varepsilon}\right) \tag{23}$$

and in the case $d > 2$, it runs in

$$\mathcal{O}\left(nm\left(\frac{n\log(nC)}{\varepsilon}\right)^{d-1}\right). \tag{24}$$

The idea was picked up by Breugem et al. (2017). They pair Martins's algorithm with the subdivision of the outcome space used in (Tsaggouris & Zaroliagis, 2007). The result is an FPTAS for MOSP problems that is worse regarding the asymptotic running time, but performs better in practice. Their asymptotic running time is (cf. Breugem et al., 2017, Theorem 3.7.)

$$\mathcal{O}\left(n^3\left(\left(\frac{n\log(nC)}{\varepsilon}\right)^{d-1}\right)^2\right). \tag{25}$$

Based, among others, on these works, (Bökler & Chimani, 2020) recently published an extensive comparison of different label ordering and selecting strategies in approximation algorithms for MOSP.

DIGRESSION: PARAMETRIZED COVERS. Note that Definition 11.2 does neither enforce the cardinality of the found cover to be minimal nor that it contains only efficient solutions. The parametrization of approximate sets of efficient solutions w.r.t. their *cardinality*, the *spacing* between the solutions, and other parameters is a research field beyond the scope of this chapter. A good overview on the topic can be read in Audet et al. (2021). The authors were not aware of the publication Bazgan et al. (2017) which in our opinion contains crucial theoretical results for the further development of algorithms in the field. The authors parameterize covers using spacing, cardinality, and coverage (i.e., how many efficient solutions are α-covered by an approximate solution). They study the existence of covers that fulfill bounds imposed on the three parameters mentioned in this paragraph. In the biobjective case they exist and generic algorithms for their generation are proposed. For the general multiobjective case, such covers might not exist but if they do, the bound on the cardinality must be raised. While the FPTAS discussed in this section are best possible approximations regarding quality loss w.r.t. the computation of minimal complete sets of efficient solutions, it shall be noted that purposefully parametrized approximations using cardinality, spacing, and coverage seem to be more relevant in practice.

OUTLINE AND CONTRIBUTIONS In Section 11.3 we discuss the outcome space partition for MOSP used in Tsaggouris and Zaroliagis (2007). Section 11.4 contains our main contribution in this chapter: we pair the the outcome space partition with the MDA and get a new FPTAS for MOSP, the *MD-FPTAS*. For MOSP instances with constant arc costs the new FPTAS works immediately. For Dyn-MOSP instance, the bounded quality loss through approximations can only be achieved stating a new assumption on the dynamic arc cost functions. Once the correctness is proven in Section 11.4.1, we analyze the asymptotic running time and space bounds of the MD-FPTAS in Section 11.4.2. In our experiments in Section 11.5 we compare the dynamic costs version of the MDA with the MD-FPTAS with regard to speed and cardinality of solutions.

11.3 OUTCOME SPACE PARTITION

For the remainder of this chapter, we consider dynamic arc cost functions from the class $\mathbb{F}$ that we define as follows.

Definition 11.3 ($\mathbb{F}$-functions)**.** Consider a continuous and piecewise linear scalar function $f : \mathbb{R}_{>0} \to \mathbb{R}_{>0}$ with $\mathcal{O}(n)$ pieces. Let each piece of f be described by the affine functions $\mathrm{aff}_i(x) = a_i x + b_i$, $i \in \{1, \dots, k\}$. Then, f is said to be an $\mathbb{F}$-*function* if $a_i \in \mathbb{R}_{\geqslant -1}$ and $b_i \in \mathbb{R}_{\geqslant 0}$ for every $i \in \{1, \dots, i\}$.

Note that because if the abuse of notation announced in Remark 10.1 regarding the input of dynamic arc cost functions, these functions can be $\mathbb{F}$-functions for fixed arcs $a \in A$. In fact, constant arc cost functions and piecewise linear FIFO functions are $\mathbb{F}$ functions. The reason for the assumption $b_i \in \mathbb{R}_{\geqslant 0}$ regarding the intercepts of the affine pieces is technical and not relevant for the understanding of this section. We need this property for the proof of Lemma 11.5 that is then used to proof the correctness of the MD-FPTAS. We use the following notation.

Definition 11.4. Consider a Dyn-MOSP instance $\mathcal{I}$ with d $\mathbb{F}$-functions associated with every arc. W.l.o.g., assume that every dynamic arc cost function is defined on an interval $[0, T]$ for $T \in \mathbb{R}_{>0}$. For every $i \in \{1, \ldots, d\}$, we define the following values.

MIN. COSTS

$$c_i^{\min} := \min_{a \in A}\{\min_{\tau \in [0,T]} c_i(a, \tau)\}.$$

MAX. COSTS

$$c_i^{\max} := \max_{a \in A}\{\max_{\tau \in [0,T]} c_i(a, \tau)\}.$$

MAX./MIN. RATIO

$$C_i := \frac{c_i^{\max}}{c_i^{\min}} \text{ and } C := \max_{i \in \{1,\ldots,d\}} C_i.$$

The min. and max. costs in Definition 11.4 for an arc $a \in A$ can be computed in $\mathcal{O}(n)$ time. Moreover, by Assumption 11.1, C_i is well defined since we consider only positive arc costs in this chapter. The next definition and the two lemmas in this section can also be found in (Breugem et al., 2017; Tsaggouris & Zaroliagis, 2007). The usage of Dyn-MOSP instances instead of MOSP instances with constant arc cost vectors does not impact the statements since they only act on the already computed costs of paths.

Definition 11.5 (Outcome Space Partition)**.** Consider a d-dimensional Dyn-MOSP instance $\mathcal{I}$ and $\varepsilon \in (0, 1]$.

PARTITION USING HYPERRECTANGLES For every node $v \in V$ we partition the outcome space of s-v-paths into disjoint axes-parallel hyperrectangles. Each hyperrectangle is represented by a d-dimensional index vector that refers to the d-dimensional generalization of its *lower left* coordinate. In dimension $i \in \{1, \ldots, d\}$ cells are indexed from 0 to $\lfloor \log_r(nC_i) \rfloor$ for $i \in \{1, \ldots, d\}$, where we set $r := (1+\varepsilon)^{\frac{1}{n-1}}$.

SIZE OF THE HYPERRECTANGLES Actually, the hyperrectangles are defined by the values of the so called **pos** *functions*. A vector $x \in \mathbb{R}^d_{\geqslant 0}$ is uniquely assigned to a cell using the $\mathbf{pos}_i$ functions, $i \in \{1, \ldots, d\}$,

$$\mathbf{pos}_i : \mathbb{R}_{\geqslant 0} \to \mathbb{N}$$
$$x \mapsto \begin{cases} 0, & \text{if } x = 0 \\ 1 + \left\lfloor \log_r \frac{x}{c_i^{\min}} \right\rfloor, & \text{else.} \end{cases}$$

pos-VALUES OF PATHS For a path p in G, we define

$$\mathbf{pos}(p, \tau_0) := \Big(\mathbf{pos}_i(c_i(p, \tau_0))\Big)_{i=1}^{d} \in \mathbb{N}^d.$$

We refer to the ith entry of $\mathbf{pos}(p, \tau_0)$ by $\mathbf{pos}_i(p, \tau_0)$, a shorthand notation for the well defined value $\mathbf{pos}_i(c_i(p, \tau_0))$.

Since we assume arc cost functions to be positive in this chapter (Assumption 11.1) the **pos**-values are well defined.

Lemma 11.1. *For any node $v \in V$ and every efficient s-v-path p, there holds*

$$\mathbf{pos}_i(p, \tau_0) \in \left\{0, \dots, 1 + \lfloor \log_r((n-1)C_i) \rfloor \right\}. \tag{26}$$

Proof. If $c(p, \tau_0) = 0$, $\mathbf{pos}(p, \tau_0) = 0$ by definition. Moreover, the **pos**-functions are monotonically increasing. Thus, it suffices to analyze the **pos**-values of an efficient path p with a possibly large cost vector. Since p is an efficient s-v-path, it is a simple path. Thus, it contains at most $(n-1)$ arcs and there holds $c_i(p, \tau_0) \leqslant (n-1)c_i^{\max}$ for every $i \in \{1, \dots, d\}$. Thus,

$$\mathbf{pos}_i(p, \tau_0) \leqslant 1 + \left\lfloor \log_r \frac{(n-1)c_i^{\max}}{c_i^{\min}} \right\rfloor = 1 + \lfloor \log_r((n-1)C_i) \rfloor$$

which proves the statement. □

After the last lemma, we observe that the **pos** function assigns paths to a cell in our outcome space partition and that every cell can be reached.

Lemma 11.2. *The number of permanent paths in a Dyn-MOSP algorithm that, for every $v \in V$, stores at most one s-v-path in every cell as defined in Definition 11.5, is at most*

$$n \left\lfloor \prod_{i=1}^{d} \frac{n-1}{\varepsilon} \ln(nC_i) \right\rfloor \in \mathcal{O}\left(n \left(\frac{n}{\varepsilon} \ln(nC)\right)^d\right). \tag{27}$$

Proof. Proving the bound is immediate using Assumption 11.2 and basic logarithmic rules. Then, using $\ln(1+\varepsilon) \in \Theta(\varepsilon)$, we can write $\log_r(nC_i)$ as $\frac{n-1}{\varepsilon} \ln(nC_i)$ for every $i \in \{1, \dots, d\}$. □

Note that, (27) is polynomial the input size of a Dyn-MOSP instance and in ε^{-1}. This means that upon proving that this would cover every efficient path with an approximation factor of at most $(1+\varepsilon)$, an algorithm that outputs one path per hyperrectangle is an Dyn-MOSP FPTAS. This outcome space partitioning approach does not guarantee that the output paths are efficient. The Dyn-MOSP FPTAS introduced in (Tsaggouris & Zaroliagis, 2007) indeed indexes paths according to their **pos**-values and returns at most one path per hyperrectangle, possibly filling all of them. The FPTAS from Breugem et al. (2017) follows a different approach and uses $\preceq_D$-checks on the paths **pos**-values to keep at most one path per hyperrectangle if its coordinates are not dominated by the **pos**-values of other paths. Still, it could be that a dominated path is stored in an hyperrectangle and then returned. We discuss both approaches in the next section in conjunction with the MDA. The approach from (Tsaggouris & Zaroliagis, 2007) does not require $\preceq_D$-checks and is useful to achieve state of the art asymptotic running time. In practice, the approach from (Breugem et al., 2017) is superior since it stores and thus expands less paths.

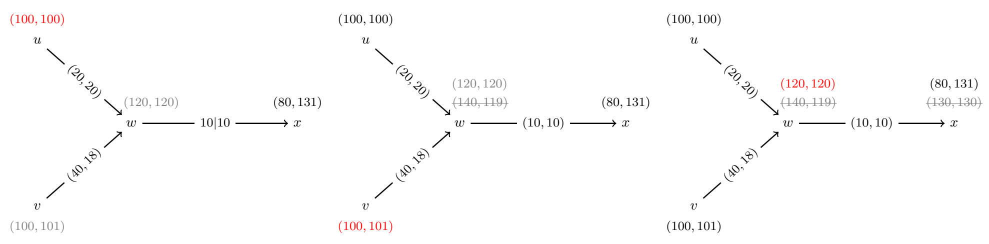


Figure 25: Corresponds to Example 11.3. Three consecutive iterations of the MD-FPTAS. The costs of the extracted label p^* in every iteration is marked in red, the permanent labels $p \in \widetilde{P}_{sv}(\tau_0)$ in black, and the explored paths generated in propagate or nextQueuePath in grey.

11.4 MULTIOBJECTIVE DIJKSTRA FPTAS

We can describe our new FPTAS for Dyn-MOSP, the MD-FPTAS, in one sentence: Use the MDA, sort the paths in the priority queue Q in lex. non-decreasing order w.r.t. their **pos**-values, and determine dominance in Line 3 of propagate and in Line 7 of nextQueuePath also using the paths' **pos**-values. In other words, we replace the role of the original arc cost functions $c(p, \tau)$ for a path p and a state τ with the **pos** functions. Only for the expansions of an s-v-path p along an outgoing arc $a \in \delta^+(v)$ we still use $c(p \circ a, \tau_0) = c(p, \tau_0) + c(a, c(p, \tau_0))$ to calculate the exact costs of the resulting path. We refer to the original publication (Maristany de las Casas, Borndörfer, et al., 2021) which contains a pseudocode of the MD-FPTAS. Since the changes to the original MDA are as marginal as stated in this paragraph, we do not include the pseudocode in the thesis explicitly.

Following convention in the literature, we use P_{sv}, $P^*_{sv}(\tau_0)$, and $\widetilde{P}_{sv}(\tau_0)$ for the set of s-v-paths in an input graph G, a minimal complete set of efficient s-v-paths with initial state τ_0, and a cover of the set of s-v-paths with initial state τ_0, respectively. $\widetilde{P}_{sv}(\tau_0)$ is of course dependent on a $\varepsilon > 0$ that is always clear from the context in the remainder of this chapter.

Example 11.3 (Dominance and Path Discarding in the MD-FPTAS)**.** In this example we deal with a MOSP instance with two constant arc cost functions per arc. Figure 25 visualizes the situation *at the end* of each of three subsequent iterations of the MD-FPTAS. The illustrated graph is a subgraph of a larger graph with $n = 10$ nodes and $c_i^{\min} = 1$ for $i \in \{1, 2\}$. We set $\varepsilon = 0.5$. Cost vectors in black next to a node in 25 symbolize the cost vector of a path that has already been made permanent. For example, in the first depicted iteration, the leftmost one, a path ending at node x with costs $(80, 131)$ exists. Similarly, cost vectors in gray symbolize existing explored paths. The red cost vector in every iteration represents the explored path that is extracted from the priority queue of the MD-FPTAS.

In the **first and leftmost iteration**, s-u path with costs $(100, 100)$ is extracted from Q and during propagate in the MD-FPTAS the s-w-path with costs $(120, 120)$ at node w is inserted into Q. In the **second iteration**, the s-v-path with costs $(101, 100)$ is extracted from Q and during propagate in the MD-FPTAS the new s-w-path with costs $(140, 119)$ is explored. Since the s-w-path with costs $(120, 120)$ is lex. smaller w.r.t. the **pos**-values, the new path is, for now, not stored in Q after the check in Line 5.

In the **third iteration**, the s-w-path with costs $(120, 120)$ at node w is extracted from Q. When nextQueuePath is called, the extension of the permanent s-v-path with costs $(100, 101)$ along the arc (v, w) is repeated to obtain, again, the path with costs $(140, 119)$. In the exact scenario this path is not dominated by the extracted s-w-path with costs $(120, 120)$. However, in the MD-FPTAS, more precisely in a call to nextQueuePath, it is rejected since $107 = \mathbf{pos}(120) < \mathbf{pos}(140) = 110$ and $\mathbf{pos}(120) = \mathbf{pos}(119) = 107$. The iteration continues and the extracted path is expanded along the arc (w, x). The resulting s-x-path with costs $(130, 130)$ is also rejected despite it would be made permanent in the exact scenario: there holds $\mathbf{pos}(80) < \mathbf{pos}(130)$ and $\mathbf{pos}(130) = \mathbf{pos}(131) = 109$.

11.4.1 Correctness of the mdFPTAS

Our exposition relies on the correctness of the MDA for Dyn-MOSP problems. Since the **pos** function is monotonically non-decreasing and its input are the paths' costs, the lexicographic ordering in Q w.r.t. the paths' **pos**-values is, up to **pos**-value equality, equivalent to the lexicographic ordering of the paths w.r.t. (17). Similarly, for two paths p, q with $c(p, \tau_0) \preceq_D c(q, \tau_0)$ we have either $\mathbf{pos}(p, \tau_0) = \mathbf{pos}(q, \tau_0)$ or $\mathbf{pos}(p, \tau_0) \preceq_D \mathbf{pos}(q, \tau_0)$. Then, the MD-FPTAS works analogously to the MDA but it sorts paths in the algorithm's priority queue and it performs dominance tests using the **pos**-values of the considered paths. This immediately gives us the following result.

Lemma 11.3. *Consider a Dyn-MOSP instance $\mathcal{I}$ and an $\varepsilon \in (0, 1]$. For any $v \in V$ let $\widetilde{P}_{sv}(\tau_0)$ be the set of paths returned by the MD-FPTAS. If an s-v-path p is discarded, there is a permanent s-v-path $q \in \widetilde{P}_{sv}(\tau_0)$ that dominates p or is equivalent to p w.r.t. the paths'* **pos***-values.*

To prove that the MD-FPTAS is indeed an FPTAS for MOSP, we need to show that for every $v \in V$, the set $\widetilde{P}_{sv}(\tau_0)$ of s-v-paths returned by the MD-FPTAS is a $(1 + \varepsilon)$-cover of the set of s-v-paths in G. The following lemma ensures that discarded paths in the MD-FPTAS are $(1 + \varepsilon)$-covered by a path in the algorithm's output.

Lemma 11.4 ((cf. Breugem et al., 2017, Lemma 3.5.)). *Consider two s-v-paths p and q for some $v \in V$ and assume $\mathbf{pos}(p, \tau_0) \preceq_D \mathbf{pos}(q, \tau_0)$, then p r-covers q for $r = (1 + \varepsilon)^{\frac{1}{n-1}}$.*

Proof. Using basic logarithmic rules and the fact that $\lfloor x \rfloor \leqslant \lfloor y \rfloor \Rightarrow x - 1 \leqslant y$ for any $x, y \in \mathbb{R}$, we obtain, for any $i \in \{1, \ldots, d\}$,

$$\mathbf{pos}_i(p, \tau_0) \leqslant \mathbf{pos}_i(q, \tau_0) \Rightarrow \log_r(c_i(p, \tau_0)) - 1 \leqslant \log_r(c_i(q, \tau_0)).$$

From the r.h.s of the implication, we obtain $c_i(p, \tau_0) \leqslant r c_i(q, \tau_0)$ for any $i \in \{1, \ldots, d\}$ which proves the statement. $\square$

Note that $r = (1 + \varepsilon)^{\frac{1}{n-1}}$, is greater than 1 and smaller than $(1 + \varepsilon)$. Thus if for every efficient s-v-path q the MD-FPTAS would return an s-v-path p with $\mathbf{pos}(p, \tau_0) \preceq_D \mathbf{pos}(q, \tau_0)$, we would have an r-cover of the set of s-v-paths and we would be done. However, our dynamic programming approach expands s-v-paths arc by arc and the provable bound on the approximation factor turns out to be worse after every arc expansion.

The following technical lemma is needed to determine for what kind of dynamic arc cost functions the MD-FPTAS finds proper covers.

Lemma 11.5. *Let $f \in \mathbb{F}$ be a piecewise affine function with $k \in \mathbb{N}$ breakpoints. Describe the affine functions that build the pieces of f by $\operatorname{aff}_i(x) := a_i x + b_i$, $i \in \{1, \dots, k-1\}$. Moreover, let $\alpha \in \mathbb{R}_{>1}$ be a given constant. Then, for points $x, y \in \mathbb{R}_{\geqslant 0}$ with $x \leqslant \alpha y$ holds $x + f(x) \leqslant \alpha(y + f(y))$ if $b_i \geqslant 0$ for all $i \in \{1, \dots, k-1\}$.*

Proof. We consider three different cases to prove the statement.

Case 1: $f(x) \leqslant f(y)$. Since $\alpha > 1$, we have $f(x) \leqslant \alpha f(y)$. Together with $x \leqslant \alpha y$ this proves the statement.

Case 2: $x < y$ and $f(y) < f(x)$. In this case, the FIFO property of $\mathbb{F}$ functions and $\alpha > 1$ can be used to get:

$$x + f(x) \leqslant y + f(y) \leqslant \alpha y + \alpha f(y).$$

Case 3: $y < x$ and $f(y) < f(x)$. Let aff_i be the affine function with $f(y) = \operatorname{aff}_i(y)$ and aff_j the one with $f(x) = \operatorname{aff}_j(x)$. There holds $i \leqslant j$ and we define aff_l, $l \in \{i, \dots, j\}$, to be the affine function corresponding to the steepest piece of f between y and x, i.e., the one with the biggest a_l. Since $f(y) < f(x)$, it must hold that $a_l > 0$. Choosing aff_l as we do, we get $\operatorname{aff}_l(y) \leqslant f(y)$ and $f(x) \leqslant \operatorname{aff}_l(x)$. Additionally, as for any affine function with positive intercept we have $\operatorname{aff}_l(\alpha y) \leqslant \alpha(a_l y + b_l) = \alpha \operatorname{aff}_l(y)$. All in all, we can conclude

$$f(x) \leqslant \operatorname{aff}_l(x) \leqslant \operatorname{aff}_l(\alpha y) \leqslant \alpha \operatorname{aff}_l(y) \leqslant \alpha f(y).$$

Together with $x \leqslant \alpha y$ this proves the statement. □

The next theorem proves that the MD-FPTAS is indeed an FPTAS for MOSP. The proof is similar to (cf. Breugem et al., 2017, Lemma 3.5.). However, the dynamic arc cost functions considered in this section and the MDA as *baseline algorithm* make the proof interesting.

Theorem 11.1 (Error bounding)**.** *Consider a node $v \in V$ and an efficient s-v-path $p^* = (a_1, \dots, a_k)$, $k \in \mathbb{N}$. Then, the set $\widetilde{P}_{sv}(\tau_0)$ of s-v-paths returned by the MD-FPTAS contains an s-v-path $\widetilde{p}$ s.t. $c(\widetilde{p}, \tau_0) \leqslant r^k c(p^*, \tau_0)$.*

Proof. Recall that the efficiency of p^* implies that it is a simple path because the arc cost functions are non-negative and thus, $k \leqslant n - 1$. We prove the statement by induction over k, the number of arcs of p^*.

BASE CASE Consider an efficient single-arc path $p^* = (a_1)$ for $a_1 = (s, v) \in \delta^+(s)$. In the first iteration of the MD-FPTAS, p^* is added to Q during propagate. Consider the first time in which an s-v-path p is extracted from Q. If $p = p^*$ we are done, since p^* is immediately added to $\widetilde{P}_{sv}(\tau_0)$. In case $p \neq p^*$, p^* is built again during a call to nextQueuePath. If its **pos**-value is dominated by or equivalent to the **pos**-value of a path in $\widetilde{P}_{sv}(\tau_0)$, it is discarded. Otherwise, it is added to Q when it becomes the lex. smallest s-v-path candidate. Before being extracted from Q and added to $\widetilde{P}_{sv}(\tau_0)$, p^* might be replaced in Q by other s-v-paths multiple times. However, every time it becomes a candidate to re-enter Q the dominance of its **pos**-value is reassessed. We conclude that p^* is either added to $\widetilde{P}_{sv}(\tau_0)$ itself or there is an s-v-path p in $\widetilde{P}_{sv}(\tau_0)$ s.t. $\mathbf{pos}(p,\tau_0) \leqslant \mathbf{pos}(p^*,\tau_0)$. In this case, Lemma 11.4 proves the statement.

INDUCTION HYPOTHESIS We assume that

$$c(\widetilde{p},\tau_0) \leqslant r^{k-1}c(p^*,\tau_0), \tag{28}$$

holds for any $k \in \{2, \ldots n-1\}$ and efficient paths p^* with $k-1$ arcs.

INDUCTION STEP Let p^* be an efficient (s,v)-path with k arcs and let $a_k = (u,v)$ be its last arc. Due to subpath efficiency, the (s,u)-subpath p^*_u of p^* is efficient. In addition, the induction hypothesis guarantees the existence of a path $\widetilde{p}_u$ such that (28) holds for $\widetilde{p}_u$ and p^*_u. When $\widetilde{p}_u$ is extracted and made permanent, the path $\widetilde{p} = \widetilde{p}_u \circ (u,v)$ is analyzed in the MD-FPTAS version of propagate and we have

$$\begin{aligned} c(\widetilde{p},\tau_0) = c(\widetilde{p}_u,\tau_0) + c(a_k, c(\widetilde{p}_u,\tau_0)) &\overset{(28) \text{ and Lemma } 11.5}{\leqslant} \\ r^{k-1}c\left(p^*_u,\tau_0\right) + r^{k-1}c(a_k, c\left(p^*_u,\tau_0\right)) \leqslant r^{k-1}c(p^*,\tau_0). \end{aligned} \tag{29}$$

Using the same argument as the base case of this proof, we know that $\widetilde{p}$ is either added to $\widetilde{P}_{sv}(\tau_0)$ in a later iteration or it is discarded. In case $\widetilde{p}$ is added, we have $c(\widetilde{p},\tau_0) \leqslant r^{k-1}c(p^*,\tau_0) \leqslant r^{k}c(p^*,\tau_0)$ and we are done. If $\widetilde{p}$ is discarded, there is a path p in $\widetilde{P}_{sv}(\tau_0)$ such that $\mathbf{pos}(p,\tau_0) \leqslant \mathbf{pos}(\widetilde{p},\tau_0)$. By Lemma 11.4 the latter inequality implies $c(p,\tau_0) \leqslant rc(\widetilde{p},\tau_0)$. Combining this with (29), we get

$$\frac{1}{r}c(p,\tau_0) \leqslant c(\widetilde{p},\tau_0) \leqslant r^{k-1}c(p^*,\tau_0) \iff c(p,\tau_0) \leqslant r^{k}c(p^*,\tau_0),$$

which finishes the proof. □

The last equation motivates our choice of $r = (1+\varepsilon)^{\frac{1}{n-1}}$. Knowing that the path p^* in the proof is efficient, we know that it has at most $k = n-1$ arcs. Then, r^k becomes exactly $1+\varepsilon$, giving the $(1+\varepsilon)$-cover for the set P_{sv} of s-v-path in the input graph starting at s with state τ_0.

11.4.2 Space Consumption and Running Time Bounds

To ease the notation in this chapter, we set

$$T := \frac{n}{\varepsilon}\ln(nC). \tag{30}$$

The definition of T is motivated by Lemma 11.2. Our assumptions regarding the number of breakpoints of the dynamic arc cost functions and regarding the $\mathcal{O}(1)$ evaluation of analytic formulas (**pos**-functions) ensure that $c(p,\tau_0)$ and $\mathbf{pos}(p,\tau_0)$ are calculated in constant time for every $v \in V \setminus \{s\}$ and every s-v-path p when p is obtained after the expansion of its s-u-subpath along p's last arc (u, v). As always, we represent paths implicitly using labels. Each label, i.e, each path, then consumes $\mathcal{O}(1)$ memory. Thus, Lemma 11.2 already proves, that the MD-FPTAS requires $\mathcal{O}\left(nT^d\right)$ space to store the output paths. Setting $N = nT^d$ in the space consumption bound from the MDA for Dyn-MOSP instances derived in (21), we get a space complexity of

$$\mathcal{O}\left(nT^d + dnm\right) \tag{31}$$

for the MD-FPTAS.

Reducing one order of magnitude

When bounding the size of the output of their FPTASes for MOSP, the authors in Breugem et al. (2017) and Tsaggouris and Zaroliagis (2007) remark that they can be *exact in one dimension* (w.l.o.g. the last one) and thus consider a $(d-1)$-dimensional space with

$$\mathcal{O}\left(nT^{d-1}\right) \tag{32}$$

hyperrectangles. Being exact in one dimension means, in this context, that for every $(d-1)$-dimensional **pos** vector encoded in (32) an approximation algorithm can keep a solution with these $(d-1)$ cost entries and the best possible value in the d^{th} dimension. Given a path's p label, we assume that we can access $\mathbf{pos}(p, \tau_0)$ in constant time. If there is a path p' in this hyperrectangle and $c_d(p', \tau_0) > c_d(p, \tau_0)$, we replace p' with p and obtain the cover. In contrast to the other dimensions, this inequality is checked using the exact c_d cost component of the paths instead of their **pos**-values.

We remark that using a label-setting algorithm that processes paths in lex. non-decreasing order like the MD-FPTAS, we can reach the same bound (32) without differentiating between exact and approximate cost dimensions. Given the hyperrectangle defined by the d-dimensional **pos**-values, the MD-FPTAS does never output one path per cell. The reason is that we do $\preceq_D$ -checks w.r.t. the permanent paths' **pos**-values in the MD-FPTAS variants of propagate and nextQueuePath to possibly discard explored paths. It is then easy to see that the maximal size of a set of non-dominated points drawn from the vertices of the cells in the d-dimensional hyperrectangle is a $(d-1)$-dimensional diagonal. The size of this diagonal is, bounded by (32). Thus, using the definitions of $N_{\max} = \max_{v \in V} |\widetilde{P}_{sv}|$ and $N = \sum_{v \in V} |\widetilde{P}_{sv}|$ as in Section 4.4 we find

$$N_{\max} \in \mathcal{O}\left(T^{d-1}\right) \tag{33}$$

and $N \in \mathcal{O}\left(nT^{d-1}\right)$ in this section.

Bounds

The discussion in the last section allows us to improve the space bound (31) by one order of magnitude. Combining the arguments that led us to the equation (31) and the discussion from last section, we immediately obtain

Theorem 11.2 (Space Consumption of the MD-FPTAS). *The MD-FPTAS uses*

$$\mathcal{O}\left(nT^{d-1} + dnm\right) \tag{34}$$

space.

For the asymptotic running time bound, we combine (33), $N \in \mathcal{O}(nN_{\max})$, and the running time bound for the MDA derived in (20).

Theorem 11.3 (Asymptotic Running Time of the MD-FPTAS). *The MD-FPTAS runs in*

$$\mathcal{O}\left(T^{d-1}(n \log n + T^{d-1} m)\right) \tag{35}$$

for $d > 2$ *and in*

$$\mathcal{O}\left(T^{d-1}(n \log n + m)\right) \tag{36}$$

in the biobjective ($d = 2$) *scenario.*

11.4.3 Improved running time bound without dominance checks.

The MD-FPTAS as described so far uses dominance tests w.r.t. the **pos**-values of paths to determine whether explored paths can be discarded. Since dominance checks for $d > 2$ are a linear time check, this causes a quadratic dependency on T^{d-1} in the general case, i.e., for $d > 2$. However, using once again that we consider d as a constant in this chapter and that the **pos**-values are assumed to be also computed in constant time, we can get rid of the quadratic dependency.

Assume that we can compute a path's **pos**-values in $\mathcal{O}(d)$ time and that we can check in linear time w.r.t. the number of nodes in a path ($\mathcal{O}(n)$) whether a path is simple. Let p be an explored path. To obtain a variant of the MD-FPTAS with an improved running time bound, we replace the $\mathcal{O}(\mathcal{N}_{\max})$ dominance or equivalence $\preceq_D$-checks in propagate and in nextQueuePath with the $\mathcal{O}(d+n)$ time check shown in Algorithm 11.

Algorithm 11: pos dominance check

```
Input  : s-v-path $p$, s-v-paths $\widetilde{P}_{sv}(\tau_0)$ indexed by their first $(d-1)$
         pos-values.
Output : Updated $\widetilde{P}_{sv}(\tau_0)$.
1 if $p$ is a simple path then
2     if $\widetilde{P}_{sv}\left[(\mathbf{pos}_1(p,\tau_0),\dots,\mathbf{pos}_{d-1}(p,\tau_0))\right]$ is empty then
3         Store $p$ in $\widetilde{P}_{sv}$;
4     else
5         $q \leftarrow \widetilde{P}_{sv}\left[(\mathbf{pos}_1(p,\tau_0),\dots,\mathbf{pos}_{d-1}(p,\tau_0))\right]$;
6         if $c_d(p,\tau_0) < c_d(q,\tau_0)$ then
7             Replace $q$ with $p$ in $\widetilde{P}_{sv}$;
8 return $\widetilde{P}_{sv}$;
```

Using Algorithm 11 for Dyn-MOSP instances in any dimension $d > 2$ the running time of the MD-FPTAS becomes

$$\mathcal{O}\left(T^{d-1}(n \log n + m)\right).$$

The new $\mathcal{O}(d+n)$ checks on paths vanish since $\mathcal{O}(d+n) \subset \mathcal{O}\left(T^{d-1}\right)$.

Note that the asymptotic space consumption remains the same. However, using Algorithm 11 requires to store the sets $\widetilde{P}_{sv}(\tau_0)$ as arrays to allow constant time access to its elements by a path's **pos**-values. This is not needed

	$d = 2$	$d > 2$
(Tsaggouris & Zaroliagis, 2007)	$\mathcal{O}(Tnm)$	$\mathcal{O}\left(T^{d-1}nm\right)$
(Breugem et al., 2017)	$\mathcal{O}\left(T^{(d-1)}n^3\right)$	$\mathcal{O}\left(T^{2(d-1)}n^3\right)$
MD-FPTAS	$\mathcal{O}(T(n \log n + m))$	$\mathcal{O}\left(T^{d-1}(n \log n + T^{d-1}m)\right)$
MD-FPTAS with Algorithm 11	$\mathcal{O}(T(n \log n + m))$	$\mathcal{O}\left(T^{d-1}(n \log n + m)\right)$

Table 13: Overview of asymptotic running time bounds for multiple state of the art FPTAS for MOSP problems.

in the original version of the MD-FPTAS since in that version, the output sets $\widetilde{P}_{sv}(\tau_0)$ can grow as paths made permanent. Thus, in practice, the MD-FPTAS paired with Algorithm 11 is required to allocate much more space from the beginning.

11.4.4 Final theoretical observations

Table 13 contains an overview of the asymptotic running time bounds of the state of the art FPTAS for MOSP that were introduced in this section and mentioned in this chapter's introduction.

We observe that the MD-FPTAS using Algorithm 11 instead of linear time dominance checks achieves the best bounds in every dimension. As mentioned earlier, it consumes a prohibitive amount of memory in practice. Thus, the originally discussed version of the MD-FPTAS that generates the sets $\widetilde{P}_{sv}(\tau_0)$ iteratively is used in Section 11.5 for our experiments.

Remark 11.1 (FPTAS for One-to-One MOSP)**.** *Note that converting to convert the MDA into an FPTAS for MOSP we only needed to introduced the* **pos**-*values to check dominance between paths. Thus, it is immediate to see that the T-MDA can also be adapted to obtain an FPTAS for the One-to-One MOSP problem. The resulting FPTAS has the same running time bound as the variants of the MD-FPTAS discussed so far in this chapter.*

11.5 EXPERIMENTS

The MD-FPTAS inherits the behavior of the MDA and the T-MDA when compared with the Martins's algorithm and the NAMOA$^*_{dr}$-lazy algorithm. Originally in (Maristany de las Casas, Borndörfer, et al., 2021) we compared it to the FPTAS introduced in (Breugem et al., 2017). The output of both FPTASes coincides, so back then we were interested in comparing both algorithms' running time. This comparison is no longer relevant within the scope of this thesis. Thus, in this section, we shorten the experimental results reported in (Maristany de las Casas, Borndörfer, et al., 2021). We focus only on the quality of the achieved approximations.

The following two reasons justify our decision.

- Originally, we compared the *Hybrid FPTAS* from (Breugem et al., 2017) with the MD-FPTAS. However, since the Hybrid FPTAS is an implementation of Martins's algorithm performing dominance checks using the paths' **pos**-values, the running time comparison resembles the results obtained and discussed in prior chapters of this thesis.

- Originally, we considered Grid graphs and NetMaker graphs with two and three-dimensional *constant and integer* arc cost functions as in prior sections of this thesis. However, as discussed in (Maristany de las Casas, Borndörfer, et al., 2021) the **pos** functions on these instances cluster the outcome space in too many cells. As a consequence, no two paths share a cell and thus, the output of the FPTASes coincides with the output of the exact algorithms. This in turn causes the **pos** calculations to be a useless computational overhead and hence, the FPTASes are slower than their exact counterpart.

A meaningful experimental setting for this chapter is to consider once again the EXP instances from (Hansen, 1980) and instances of the *Horizontal Flight Planning Problem* first defined in (Blanco et al., 2016). These instances were originally single-criterion Time Dependent Shortest Path instances and we expand them here to Dyn-MOSP scenario, and made available by our industry partner Lufthansa Systems GmbH.

Static Biobjective Exponential Instances

We use the EXP instances by Hansen (1980) in their original form, i.e., with bidimensional arc cost vectors. Recall that on these instances, all paths are efficient (see Figure 2). We use the EXP instances defined on graphs with at leasto 19 nodes. Instances defined on smaller graphs are solved in less than a millisecond and are thus not relevant. We are interested in the number of paths that are needed to compute a ε-cover $\widetilde{P}_{st}$ for these instances.

Dynamic Biobjective Instances – Horizontal Flight Planning

The Dyn-MOSP instances motivated by the Horizontal Flight Planning (HFP) problem introduced in (Blanco et al., 2016; Blanco et al., 2017). The digraph in this instance has 410387 nodes and 878902 arcs and is called Airway Network. The arcs are the direct connections between predefined coordinates (the graph's nodes) along which commercial aircraft are allowed to fly. On www.skyvector.com an Airway Network can be displayed. We define two dynamic arc cost functions.

The first one encodes the flight duration. I.e., the duration of the traversal of an arc depending on the time point at which the tail of the arc is reached. The duration is influenced by weather conditions. A weather prognosis estimates wind for 30h. The information available to us is discrete and contains an updated wind vector for every arc every 3h. Thus, we have 10 data points per arc between which we interpolate wind linearly (Blanco et al., 2016, cf.).

The second function models the aircraft's fuel consumption along an arc depending on the aircraft's weight at the arc's tail node. In our model we get 171 initial weights per arc and the corresponding consumption for each weight. The difference between two consecutive weights is 500kg. Also in the weight functions, data points are interpolated linearly.

The single pieces of the duration function can have positive or negative slopes depending on the wind but the FIFO property still holds as shown in (Blanco et al., 2016). The consumption function yields an always positive slope since clearly, a higher initial weight will cause a higher consumption. It is therefore also FIFO and, more importantly, the intercepts of its affine pieces are positive, hence fulfilling the requirements from Theorem 11.1.

In this setting, we define One-to-All Dyn-MOSP instances. For the source nodes, we choose all 586 large passenger airports[1] contained in the data made available to us by Lufthansa Systems.

In a fully realistic setting the Flight Planning (FP) problem rather than the Horizontal Flight Planning problem from this chapter is solved. The FP problem is defined in more detail in (Blanco et al., 2022). The main difference is that the FP problem considers a layered graph with around 40 layers. Every layer is essentially a copy of the graph used in the HFP problem and represents the Airway Network at a fixed altitude called a Flight Level. An aircraft can change its layer via so called *climb* and *descend* arcs. One can imagine that slight variations on the aircraft weight when a climb or a descend is started have a big impact on the aircraft's weight at the head node of the arc as well as on the duration needed to traverse the arc. In our experiments with the 3D Flight Planning solver used in (Blanco et al., 2022) and closely related to Lufthansa System's route planning optimizer VOLAR, we considered bidimensional states τ encoding weight and time as optimization criteria. Out of the box, the resulting minimal complete sets of efficient paths contain multiple solutions per second for every second in an interval between the fastest possible arrival and approximately two hours later on long haul flights. Such a *resolution* is pointless in practice for decision makers and thus, good approximations of the output sets are needed in Flight Planning. Additionally, covers are interesting in this setting because in the actual Flight Planning application, expansions of labels along arcs are the bottleneck because the arc cost functions are complicated to evaluate. Thus, saving labels in our prototype translates to relevant running time savings in the production code in the application's context.

Sadly, in the One-to-One HFP instances considered in this chapter, the minimal complete sets of efficient paths are small because climbs and descends are not considered. Thus, the MD-FPTAS is not interesting for this setting. Our HFP experiments in this section were first conducted in (Maristany de las Casas, Borndörfer, et al., 2021). To get larger sets of efficient paths we decided to test the MD-FPTAS on One-to-All HFP instances. This model fits better into the scope of this thesis. It allows us to analyze the savings with regard to the number of efficient paths that are achievable using the output space clustering as defined in this chapter on a Dyn-MOSP instance with rational paths' costs.

11.5.1 Results

We used a computer with an Intel Xeon Gold 6338 @ 2.00 GHz processor. The available RAM was 36GB. All algorithms were implemented in C++ and compiled wit the GCC compiler v.7.5 with compiler optimization level

1 According to https://github.com/datasets/airport-codes/blob/master/data/airport-codes.csv as accessed on September 1st 2023.

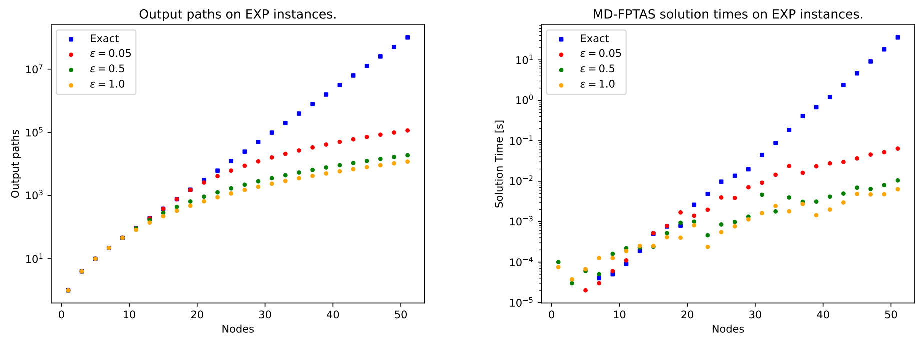


Figure 26: Visualization of the results reported in Table 14. The EXP instances were solved using the biobjective version of the MD-FPTAS. The plot on the right hand side shows the number of output paths at the rightmost node (cf. Figure 2). The plot on the left hand side shows the required time to solve the instances.

set to −O3. For the priority queues, we used our own implementation of a binary heap. We set a time limit of 5400s for all instances. Note that the results in this section were obtained using an older version of the MDA than the one used in the previous chapters of the thesis. Running times are thus not comparable to those reported in other chapters.

EXP instances

Table 14 summarizes the results obtained from the EXP instances. The instance with 19 nodes is the biggest instance in which the MDA is faster than the MD-FPTAS for $\varepsilon = 0.05$ and $\varepsilon = 0.5$. For $\varepsilon = 0.5$ the MD-FPTAS outputs 42.2% of the efficient s-t-paths calculated by the MDA. Still, it requires 0.94ms as opposed to the 0.80ms required by the MDA. The reason is that calculating the paths' **pos**-values is a non-negligible effort.

Starting with the EXP instance with 21 nodes, the calculation of **pos**-values starts to pay off for every value of ε as seen also in Figure 26. Since in EXP instances every path is efficient, the output paths returned by the MD-FPTAS are efficient. Note however, that this is not guaranteed. All in all, the relatively small number of nodes in the EXP instances and the exponential (w.r.t. the number of nodes) functions used to define the arc costs are a beneficial setting for FPTAS to reduce the output cardinality because the **pos** functions can take less values than there are efficient paths. I.e., multiple paths coincide in one cell and only one is kept.

	Exact		$\varepsilon = 0.05$		$\varepsilon = 0.5$		$\varepsilon = 1.0$	
n	$\lvert P^*_{st} \rvert$	Time [ms]	$\lvert \widetilde{P}_{st} \rvert$	Time [ms]	$\lvert \widetilde{P}_{st} \rvert$	Time [ms]	$\lvert \widetilde{P}_{st} \rvert$	Time [ms]
19	1534	0.80	1490	1.69	648	0.94	475	0.40
21	3070	2.62	2581	1.40	922	1.00	659	0.81
23	6142	4.85	4111	1.97	1264	0.46	880	0.24
25	12286	9.76	6140	3.96	1689	0.85	1164	0.55
27	24574	13.64	8758	3.85	2205	0.98	1501	0.76
29	49150	19.75	12038	7.09	2820	1.33	1892	1.14
31	98302	44.71	16053	9.18	3543	4.62	2362	1.63
33	196606	87.99	20889	14.42	4390	1.79	2886	2.43
35	393214	183.92	26624	23.66	5356	3.95	3510	1.80
37	786430	408.11	33340	16.14	6463	3.10	4184	2.73
39	1572862	676.45	41105	23.31	7722	3.13	4987	1.44
41	3145726	1205.04	50004	27.63	9124	4.15	5846	1.99
43	6291454	2381.74	60120	29.83	10694	4.95	6820	2.98
45	12582910	4643.58	71533	36.53	12452	6.92	7902	4.82
47	25165822	9132.19	84321	45.55	14393	6.44	9114	4.70
49	50331646	18195.42	98559	52.03	16538	7.99	10431	4.73
51	100663294	35892.34	114351	64.02	18873	10.46	11856	6.32

Table 14: Result collected on EXP instances with more than 19 nodes and for three different values of ε. On these instances, all paths are efficient, and thus the outputs for One-to-One and One-to-All instances are the same. The sets P_{st} report the cardinality of the solution sets at the rightmost node in every EXP graph (cf. Figure 2).

Horizontal Flight Planning Instances

As we observe in the results collected from our HFP instances and summarized in Table 15, using the MD-FPTAS is indeed a good choice for these instances. Most instances are solved by the MDA in $2^2 = 4$ to $2^6 = 64$ seconds. On these instances, the MD-FPTAS is $\times 1.23$ to $\times 1.38$ times faster than the MDA. It achieves the speedup because for the input $\varepsilon = 0.5$, the MDA outputs $\times 1.49$ to $\times 1.66$ more permanent paths. The speedup is smaller than the ratio of saved labels because calculating the paths' **pos**-values is computationally not negligible for so many paths.

The most interesting aspect of our experiments is the *a posteriori* error. We compared the exact minimal complete sets of efficient paths output by the MDA and the 1.5-covers returned by the MD-FPTAS. The reported values in these columns disregard every efficient path output in the MD-FPTAS because for these paths the approximation error is equal to zero.

Thus, we only consider output paths that are dominated and we report their deviation from efficient paths. The column % *dom. paths* in Table 15 reports the percentage of paths in the output of the MF-FPTAS that are dominated. Note that this is indeed a percentage and thus, in every group of instances, less than 1% of the output paths of the MD-FPTAS are dominated. For any $v \in V$, let $\widetilde{p} \in \widetilde{P}_{sv}(\tau_0)$ be such a dominated s-v-path, i.e., a path s.t. $c(\widetilde{p}, \tau_0) \in c(\widetilde{P}_{sv}, \tau_0) \backslash c(P^*_{sv}, \tau_0)$. We iterate over all paths in $P^*_{sv}(\tau_0)$. The path $p^* \in P^*_{sv}(\tau_0)$ that dominates $\widetilde{p}$ with the smallest euclidean distance w.r.t. the paths' costs, is used to define $\widetilde{p}$'s error. We sum the resulting (absolute and relative) errors componentwise over all nodes $v \in V$ and over all dominated s-v-paths. This way, in Table 15, we can report flight duration and fuel consumption errors separately. Then, we build the average dividing the obtained sum by the number of dominated paths in the output of the MD-FPTAS.

As hinted already, the a posteriori error is satisfactory. As noted in the last paragraph of Section 11.3, output paths can be dominated. From a practitioners point of view it is interesting to know by how much the output dominated paths deviate from efficient ones. In our HFP experiments, not only the percentage of dominated paths output by the MD-FPTAS is very low. Additionally, the relative flight duration error is less than 10^{-6} for instances that are solved in 1s to 2^7s by the MDA. On instances that take longer to be solved exactly, the MD-FPTAS makes a relative error in the flight duration of at most 0.84×10^{-6}s. The relative error for the fuel consumption, denoted as *Weight* in Table 15 behaves similarly but even better. In absolute terms, the MD-FPTAS does an average error of at most 0.84s and at most 3×10^{-6}kg.

All in all we observe than in contrast to the results obtained in (Maristany de las Casas, Borndörfer, et al., 2021) for MOSP instances with constant arc cost functions, Dyn-MOSP instances can benefit from the MD-FPTAS since less paths are made permanent. The a posteriori error is much better than the $(1 + \varepsilon)$ bound chosen in our experiments. The main reason is that the error increases with every arc in an efficient path. Thus, the outcome space partition from Definition 11.5 is chosen s.t. the longest (w.r.t. the number of arcs) possible simple path is still covered. Such a path hast $n - 1$ arcs but in our examples there are no paths with nearly as many arcs. Thus, the hyperrectangles defined by the **pos**-values of the efficient paths in the HFP instances are much smaller than for a path with $(n-1)$ arcs in the Airway Network. As a result, storing one path per cell in this *region* of the outcome space partition gives a much better approximation as the one specified by the a priori bound.

Solution time	Inst's	Time [s]			Permanent paths			A posteriori error				
									$c_1 \hat{=}$ Duration		$c_2 \hat{=}$ Weight	
		MDA	MD-FPTAS	Speedup	MDA	MD-FPTAS	$\frac{\text{Exact}}{\text{Approx.}}$	dom. paths	Abs. [s]	Rel. $[\times 10^{-6}]$	Abs. [kg]	Rel. $[\times 10^{-6}]$
$[0, 1)$	6	0.57	0.50	1.14	705151.09	472975.54	1.49	0.54	0.15	3.5	0.04	0.0
$[2^0, 2^1)$	27	1.33	1.06	1.25	1777373.25	1107720.18	1.60	0.64	0.15	0.0	0.07	0.0
$[2^1, 2^2)$	57	2.92	2.23	1.31	3673247.29	2207167.77	1.66	0.93	0.22	0.0	0.07	0.0
$[2^2, 2^3)$	137	5.80	4.71	1.23	7237552.34	4852894.94	1.49	0.55	0.20	0.0	0.10	0.0
$[2^3, 2^4)$	151	11.15	9.05	1.23	12011827.36	8162261.12	1.47	0.61	0.16	0.0	0.11	0.0
$[2^4, 2^5)$	112	22.30	17.47	1.28	19546100.32	13055535.00	1.50	0.65	0.23	0.0	0.10	0.0
$[2^5, 2^6)$	68	41.39	29.89	1.38	27680969.25	16725621.46	1.66	0.97	0.21	0.0	0.10	0.0
$[2^6, 2^7)$	18	91.81	62.96	1.46	23214889.22	15330571.45	1.51	0.69	0.19	0.0	0.16	0.0
$[2^7, 2^8)$	9	174.92	139.42	1.25	30943964.99	21951139.08	1.41	0.40	0.35	8.0	0.11	0.0
$[2^8, 2^9)$	1	290.97	132.18	2.20	55580871.00	18511900.00	3.00	0.48	0.84	17.0	0.23	3.0

Table 15: Results obtained from our 586 HFP instances. The instances are grouped depending on the solution time needed by the MDA to solve them. The intervals used to group the instances are shown in the column *Solution time*. The *Inst's* column denotes the number of instances encoded in the corresponding row of the table. The a posteriori relative error, denoted by ε, is calculated with an accuracy of 10^{-6}. Zeros in this columns thus mean that the error is smaller than the used accuracy. All reported averages are geometric means.

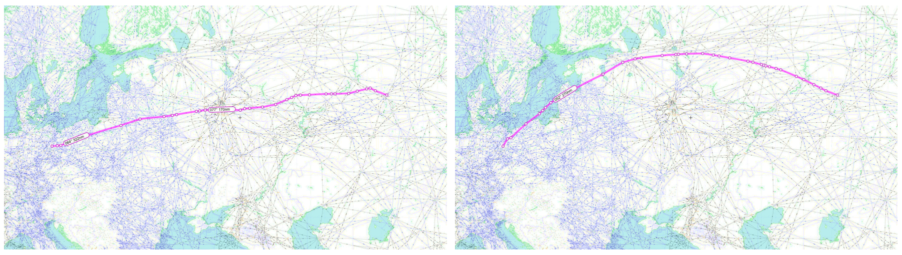


Figure 27: Least consumption (left) and fastest (right) routes from Berlin to Yekaterinburg. Source: www.skyvector.com.

Figure 27 ends this section. It contains two screenshots of routes from Berlin to Yekaterinburg. The trajectories are meant to show that indeed optimizing flights w.r.t. fuel consumption and w.r.t. time can lead to very different routes. This fact is well known in the 3D Flight Planning problem and as shown in Figure 27 also needs to be considered in HFP instances. The choice of the s-t-pair used for the figure is a relict of the time when we published (Maristany de las Casas, Borndörfer, et al., 2021). Back then the student assistant Arno Kühner wrote the export of the routes computed in the MD-FPTAS to www.skyvector.com and he confessed to always have wanted to visit Yekaterinburg.

11.6 CONCLUSION AND OUTLOOK

Good approximations for MOSP and Dyn-MOSP instances are the key for a successful transition of this research area from theory to practice. The need for approximations arises from the inherent intractability of the solution sets in Multiobjective Combinatorial Optimization problems. Of course, correlated objective functions help to reduce the expected size of the solution sets. But as we have seen in this chapter, the arguably correlated duration and fuel consumption of a flight can still cause an unnecessary (from a practitioner's point of view) amount of efficient solution paths. Having FPTAS for MOSP and Dyn-MOSP is great as they constitute the best possible class of approximations if the a priori error bound is the main concern. However, as mentioned in the introduction to this chapter, an approximation algorithm in which a decision maker can parametrize the solution set in terms of spacing, cardinality, and coverage (cf. Bazgan et al., 2017) may be better suited. The presented MD-FPTAS has state of the art asymptotic running time and memory consumption bounds. It also works well in practice when tested on Dyn-MOSP instances like the HFP instances from this chapter. Having devoted Part iii of this thesis to bring MOSP closer to practice, we see the development of approximations and the transfer of preprocessing techniques for single-criterion Shortest Path problems to the multiobjective scenario as the next steps to take researchwise to make MOSP an appealing modeling technique in practice.

Part IV

THE MULTIOBJECTIVE DIJKSTRA ALGORITHM AS A SUBROUTINE

12 INTRODUCTION

In the third and last part of the thesis, we consider static arc cost functions. Other than this, the next two chapter have little in common. This is why this introduction is rather short. The subsequent chapters have standalone introductions. Our goal was to find out how we can use the MDA to solve other problems than the MOSP problem. In other words, we were looking for optimization problems in which multiobjective shortest paths are computed in a subroutine. We came up with two ideas that are unrelated with each other and also use the MDA in very different ways.

The first idea is that Dynamic Programming algorithms for Discrete Multiobjective Optimization (DMO) problems often solve a MOSP problem implicitly. On a very high level we can say that DMO problems can be solved with Dynamic Programming algorithms if they fulfill a Bellman equation similar to the one stated in Theorem 3.1 for MOSP problems. For $i \in \mathbb{N}$, a Bellman equation for a DMO states how to derive efficient solutions for a subproblem of size $i+1$ from efficient solutions of size i.

An example is the *Multiobjective Minimum Spanning Tree* (MO-MST) problem. In this multiobjective variant of the Minimum Spanning Tree problem, greedy algorithms do not perform well and spanning trees of subgraphs of cardinality $i < n$ can be can be expanded using cut edges that *connect* the spanned subgraph with a subgraph of cardinality $i+1$. Chapter 13 is devoted to this problem.

In Chapter 14 we consider the k-*Shortest Simple Path* problem. In a black box algorithm from Roditty and Zwick (2012) we can solve their core subroutine, the *Second Shortest Simple Path* problem, using a slightly modified version of the biobjective version of the MDA. Despite the intractability of the Biobjective Shortest Path (BOSP) problem, the new bidimensional arc cost function defined in our approach is such that we can guarantee a polynomial output size w.r.t. the size of the input graph. Indeed, even using a biobjective subroutine, the new algorithm achieves an asymptotic running time bound that is on par with the state of the art.

13 MULTIOBJECTIVE MINIMUM SPANNING TREE PROBLEM

We consider the *Multiobjective Minimum Spanning Tree* (MO-MST) problem. The problem generalizes the well known Minimum Spanning Tree (MST) problem to the case in which each edge of the input *undirected* graph is weighted by a vector of costs rather than a scalar. Then, the multidimensional costs of a tree are the sum of the costs of the edges in the trees. While in MO-MST problems the set of feasible solutions remains the same as in the MST, the notion of optimal trees varies and different solution approaches are required to solve the problem to optimality. *Efficient spanning trees* are defined similarly to efficient paths in the previous chapters of this thesis: A spanning tree t is efficient if no spanning tree exists in the input graph s.t. its vector of costs is at least as good as t's in every dimension and better than t's in at lest one dimension. MST instances are usually solved using greedy algorithms like Prim's and Kruskal's (Kruskal, 1956; Prim, 1957). For MO-MST instances such algorithms are not suitable. In a nutshell, the problem is that greedy algorithms enlarge trees choosing a *best possible* new edge. This notion requires a total order of the edges according to their cost. However, in a scenario with arc cost vectors, multiple total orders exist. A detailed discussion on this topic can be read in (Ehrgott, 2005, Section 9.2.). This observation opens up the long lasting search for algorithmic approaches to solve MO-MST problems.

13.0.1 Literature Review and Outline

The MO-MST problem is well studied in the literature. For a good introduction to the topic with multiple insights on different solution approaches, we refer to (Ehrgott, 2005, Section 9.2.). The recent survey by Fernandes et al. (2020) benchmarks different algorithms on a huge set of instances providing good insights on what techniques work best in practice nowadays.

The search for algorithmic approaches other than greedy algorithms for MO-MST problems leads to different answers depending on the problem's dimension. In the biobjective case, two phase algorithms (Hamacher & Ruhe, 1994; Ramos et al., 1998; Steiner & Radzik, 2008; Sourd & Spanjaard, 2008; Amorosi & Puerto, 2022) have been proven to be efficient with the work by Sourd and Spanjaard (2008) being the indisputable state of the art. These algorithms compute the supported efficient spanning trees in the first phase and the unsupported efficient solutions in the second. Supported solutions are obtained solving multiple instances of the MST problem via scalarization of the objective functions. For the computation of unsupported solutions, different techniques are used. In their state of the art algorithm Sourd and Spanjaard (2008) split the second phase into two subphases. First they find unsupported solutions performing neighborhood search starting from the already found efficient trees. Finally the missing efficient trees are found solving multiple MIPs that use the costs of the existing efficient trees as bounds. The reason for the superb performance of their algorithm is that in almost every practical instance, all efficient trees are already computed after the neighborhood search.

For edge costs with more than two dimensions, efficient two phase approaches are not known and hard to design as discussed for example in (Ehrgott & Gandibleux, 2000). This once again opens up the search for efficient MO-MST algorithms and leads to *Dynamic Programming* algorithms. In this context, two publications by Di Puglia Pugliese, Guerreiro, and Santos stand out. They first published a Dynamic Programming algorithm for the MO-MST problem in (Di Puglia Pugliese et al., 2014) and afterwards they notably improved their approach in (Santos et al., 2018) designing the *Built Network (BN) algorithm*. In it they include an elegant way of ensuring that every dynamically built tree is only considered once. By doing so, they solve an arising *symmetry* issue in their original algorithm. Few other algorithms for MO-MST instances with $d > 2$ dimensions are known in the literature. In the PhD thesis of Lacour (2014) a generic hybrid algorithm for Multiobjective Combinatorial Optimization problems is applied to MO-MST. It combines a ranking approach to generate some *supported solutions* with a branch and bound approach to complete the set of efficient solutions. In a sense, Lacour's algorithm can be seen as a two phase approach. However, applying his algorithm to MO-MST instances is a marginal part of his thesis without many details. This makes re-implementing and retracing his results a challenging task. Other references already considered in (Fernandes et al., 2020) like (Corley, 1985; Perny & Spanjaard, 2005) determine a set of solutions that is guaranteed to contain a correct output. The algorithm presented in (Hamacher & Ruhe, 1994) enhances the algorithm presented in (Corley, 1985) but it still can output trees that are not efficient.

OUTLINE Section 13.1 begins with the formal definition of the MO-MST problem. Then, in Section 13.2 we embed the MO-MST problem in a Dynamic Programming context. Afterwards and similar to (Santos et al., 2018, Section 3.1, Section 3.2), we discuss how to transform a MO-MST instance into an instance of the One-to-One Multiobjective Shortest Path (MOSP) problem defined on a so called transition graph $\mathcal{G}$. Both instances' solution sets are equivalent. In Section 13.2.2 to Section 13.2.5 we discuss our first main contributions: how to reduce the size of $\mathcal{G}$. The resulting One-to-One MOSP instance defined on the reduced transition graph is used in our second main contribution: the *Implicit Graph Multiobjective Dijkstra Algorithm* (IG-MDA) introduced in Section 13.3. It takes advantage of the MOSP-related expertise discussed in prior chapter of this thesis. To build a meaningful experimental setting, we also use these techniques, whenever possible, in our implementation of the BN algorithm. The resulting variant of the original algorithm from (Santos et al., 2018) is described in 13.4. In Section 13.5 we present the results of our computational experiments in which we benchmark the IG-MDA against our version of the BN algorithm. Since on bi-dimensional MO-MST instances both algorithms are clearly outperformed by the two phase algorithm from Sourd and Spanjaard (2008), we use three and four-dimensional instances in our experiments. We consider all such MO-MST instances from (Santos et al., 2018) and a big subset of the instances used in (Fernandes et al., 2020). Thereby, we achieve the last contribution in this paper: to the best of our knowledge the size of the solved instances is bigger than so far in the literature.

13.1 MULTIOBJECTIVE MINIMUM SPANNING TREE PROBLEM

We proceed wit the formal definition of the MO-MST problem. An instance of the problem is a tuple (G, s, d, c) consisting of an undirected connected graph $G = (V, E)$, a d-dimensional edge cost functions $c : E \to \mathbb{R}^d_{\geqslant 0}$, and a node $s \in V$ that is assumed, w.l.o.g, to be the root of every spanning tree in G. We characterize a tree t in G by its edges, i.e., $t \subseteq E$. Then, the costs of t are defined as $c(t) := \sum_{e \in t} c(e) \in \mathbb{R}^d_{\geqslant 0}$ and we refer to the set of nodes spanned by t by $V(t) \subseteq V$. For a set T of trees, we set $c(T) := \{c(t) \mid t \in T\}$. For a subset $U \subseteq V$ of nodes in G, we write $G(U)$ to refer to the subgraph of G induced by U. Similarly, we denote the subgraph of G spanned by t as $G(t) := G(V(t))$.

Definition 13.1 (Efficient Trees)**.** Consider a MO-MST instance $\mathcal{I} = (G, s, d, c)$. Let t, t' be trees in G. t *dominates* t' if $V(t) = V(t')$, $c(t) \leqslant c(t')$, and $c(t) \neq c(t')$. Moreover, t is an *efficient spanning tree* of $G(t)$ if it is not dominated by any other spanning tree of $G(t)$. The cost vector of an efficient spanning tree of G is called an *nondominated cost vector* of $\mathcal{I}$.

We are interested in computing *minimal complete sets of efficient trees*: a subset of the efficient spanning trees of a given graph G w.r.t. an edge cost function c s.t. for every nondominated cost vector, there is exactly one efficient tree in the subset. We can now define the MO-MST problem formally.

Definition 13.2 (Multiobjective Minimum Spanning Tree Problem)**.** Consider an undirected graph $G = (V, E)$, a root node $s \in V$, and d-dimensional edge cost functions $c : E \to \mathbb{R}^d_{\geqslant 0}$. Let T be the set of all spanning trees of G. The *Multiobjective Minimum Spanning Tree* (MO-MST) problem is to find a minimal complete set $T^* \subseteq T$ of efficient trees. We refer to the pair (G, s, d, c) as a d-*dimensional MO-MST instance*.

On a complete graph with $n \in \mathbb{N}$ labeled nodes there are n^{n-2} spanning trees (Cayley, 2009). Hamacher and Ruhe (1994) constructed bi-dimensional MO-MST instances on complete graphs in which every spanning tree has a nondominated cost vector, proving the problem's intractability. In their paper, they also prove the NP-hardness of the MO-MST problem.

13.2 DYNAMIC PROGRAMMING FOR MO-MST

In this chapter we refer to a d-dimensional One-to-One MOSP instance (cf. Definition 3.2) as tuple $(\mathcal{G}, s, t, d, \gamma)$, where $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ is a directed graph, $s \in \mathcal{V}$ the source node, $t \in \mathcal{V}$ the target node, and $\gamma : A \to \mathbb{R}^d_{\geqslant 0}$ the arc cost function. We use different symbols than so far in the thesis because in this chapter, we differentiate between the original MO-MST instance to be solved and the One-to-One MOSP instance that we actually solve to obtain a solution set that is equivalent to the one of the MO-MST instance at hand.

In Dynamic Programming, Bellman conditions are recursive expressions that state how to derive optimal/efficient solutions for a problem at hand from optimal/efficient solutions of its subproblems. Assume we are given a MO-MST instance defined on a graph $G = (V, E)$ in which all spanning trees are w.l.o.g. rooted at a node $s \in V$. Let $W \subset V$ be a set of nodes and $G(W)$ be the subgraph induced by W. In this scenario, the Bellman condition (cf. Theorem 13.1) states how to build efficient spanning trees of $G(W)$ by looking at the efficient spanning trees of the subgraphs of G induced by all subsets $U \subset W$ with $|U| = |W| - 1$.

This naturally leads to the definition of a new directed graph sometimes called the *transition graph* $\mathcal{G}$ in the Dynamic Programming literature. $\mathcal{G}$ is formally defined in Definition 13.3. For now, a high level description suffices. The nodes in $\mathcal{G}$ correspond to subsets of nodes in G. A One-to-One MOSP instance is obtained setting the node encoding $\{s\}$ as the source node and the node encoding the whole node set V of G as the target node. Then, efficient $\{s\}$-V-paths in $\mathcal{G}$ correspond to efficient spanning trees in G. We remark that the arising transition graph and One-to-One MOSP instance have been already used in the literature. In (Santos et al., 2018) the authors call the transition graph the *Built Network* but they do not derive it from the Bellman condition for MO-MST as we do. We choose to introduce the graph using Bellman conditions to stress the Dynamic Programming nature of our approach in this chapter. Recall that any node set $U \subseteq V$ induces the *cut*

$$\delta(U) := \{[u, w] \in E \mid u \in U,\ w \in V \backslash U\}$$

and we have the following basic result from graph theory.

Lemma 13.1. *Let $G = (V, E)$ be an undirected graph, t a tree in G, and $[u, w] \in E$ an edge. Then, $t' = t \circ [u, w]$ is a tree if and only if $[u, w] \in \delta(V(t))$. If that is the case, we have $|V(t)| = |V(t')| - 1$.*

In the remainder of this chapter, we superscript a set T of trees with an $*$ to refer to a minimal complete set of efficient trees in T, i.e., $T^* \subseteq T$ contains exactly one tree for every nondominated vector in $c(T)$. Given a (sub-)graph G' of G, we refer to the set of spanning trees of G' by $T_{G'}$.

Theorem 13.1 (Bellman Condition for Efficient Subtrees). *Let (G, s, d, c) be a d-dimensional MO-MST instance.*

BASE CASE *We set $t_{G(\{s\})}$ to be the spanning tree of $G(\{s\})$, a subgraph containing just the node s and no edges, and $c(t_{G(\{s\})}) = 0$. Since this is the only tree in $T_{G(\{s\})}$, we have $T_{G(\{s\})} = T^*_{G(\{s\})}$.*

RECURSION *Consider a subset $W \subseteq V$ of cardinality $k \in \{2, \dots, n\}$. A minimal complete set $T^*_{G(W)}$ of efficient spanning trees of $G(W)$ is given by*

$$\left\{ t' \circ [u, w] \,\middle|\, t' \in T^*_{G(U)},\ U \subset W,\ |W| = |U| + 1, \text{ and } [u, w] \in \delta(U) \right\}^*. \tag{37}$$

Proof. Assume there exists an efficient spanning tree t of $G(W)$ and a subset $U \subset W$ s.t. $t = t'' \circ [u, w]$ for a dominated spanning tree t'' of $G(U)$, $[u, w] \in \delta(U)$. Consider an efficient spanning tree t' of $G(U)$ that dominates t''. $t' \circ [u, w]$ is also a spanning tree of $G(W)$ (cf. Lemma 13.1) and

$$c(t') \leqslant c(t'') \Leftrightarrow c(t' \circ [u, w]) \leqslant c(t'' \circ [u, w]) = c(t).$$

Since t' dominates t'' the first inequality is strict for at least one index causing t to also be dominated. By Lemma 13.1 every spanning tree of $G(W)$ can be obtained from spanning trees in the subgraphs $G(U)$ for $U \subset W$ with $|U| + 1 = |W|$.

Then, the correctness of the statement follows by induction over the cardinality of $|W|$ and using $T_{G(\{s\})} = T^*_{G(\{s\})}$ as in the induction's base case. Note that the $*$-superscript in (37) is needed because the expansions of efficient trees of different subsets U of W can dominate each other. □

From (37) we can infer how to build the transition graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$. Its *transition nodes* $\mathcal{V}$ correspond to subsets of nodes in G. A transition node $U \in \mathcal{V}$ encoding the subset $U \subset V$ is a predecessor node of a transition node $W \in \mathcal{V}$ if W corresponds to a subset $W \subset V$ of nodes s.t. $U \subset W$ with $|W| = |U| + 1$. The arcs between both nodes are given by the edges $[u, w] \in \delta(U)$. The following definition formalizes the new graph and its arc costs. For a discrete set X, the power set of X is 2^X.

Definition 13.3 (Transition Graph)**.** Consider a MO-MST instance (G, s, d, c). We define the *transition graph of* G as the directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ with node set

$$\mathcal{V} := \left\{ V' \cup \{s\} \;\middle|\; V' \in 2^{V \setminus \{s\}} \right\}.$$

Every such node is also called a *transition node*. The outgoing arcs of node $U \in \mathcal{V}$ are induced by the cut $\delta(U)$:

$$\delta^+(U) := \left\{ (U, W) \;\middle|\; W = U \cup \{w\} \text{ if } \exists [u, w] \in \delta(U) \right\}. \tag{38}$$

The set of arcs in $\mathcal{G}$ is given by $\mathcal{A} := \bigcup_{U \in \mathcal{V}} \delta^+(U)$. If an arc $(U, W) \in \mathcal{A}$ is induced by an edge $[u, w] \in E$, we call $[u, w]$ the *preimage edge* of (U, W) and write $[u, w] = (U, W)^{-1}$. Moreover, we refer to (U, W) as a *copy of* $[u, w]$ *in* $\mathcal{G}$. We define a d-dimensional arc cost function $\gamma : \mathcal{A} \to \mathbb{R}^d_{\geqslant 0}$ setting $\gamma(a) = c(a^{-1})$.

In Definition 13.3 and in the remainder of the paper, we incur in a slight abuse of notation when we denote a transition node U and simultaneously mean the transition node in $\mathcal{V}$ and the subset $U \subseteq V$ of nodes in the original graph G. Note that for an arc (U, W) as defined in (38), we have $U, W \in \mathcal{V}$ and $U \cup \{w\} = W$ for some node $w \in V$, i.e., W is an expansion of U by just one node. Even though we defined the set $\mathcal{A}$ using the nodes' outgoing arcs, it of course also allows us to access sets $\delta^-(U)$ of incoming arcs for every transition node U. The following remark is important for the understanding of the remainder of this section.

Remark 13.1. *The transition graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ as defined in Definition 13.3 is a layered acyclic multigraph.*

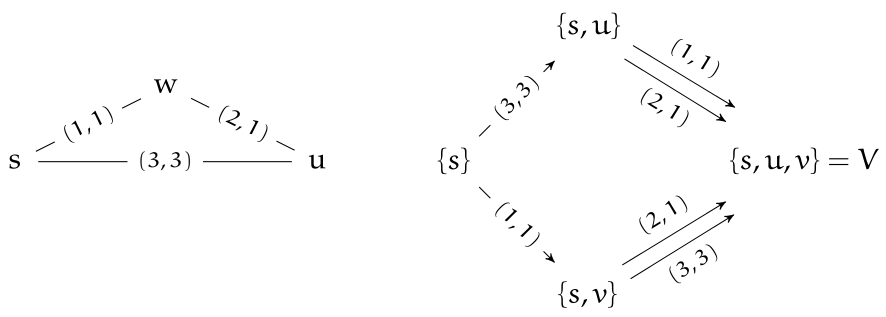


Figure 28: Left: 2d MO-MST instance (G, s, d, c). Right: Corresponding One-to-One MOSP instance $(\mathcal{G}, \gamma, \{s\}, V)$

LAYERED *For* $k \in \{1, \cdots, n\}$ *the* k^{th} *layer of* $\mathcal{G}$ *consists of all nodes in* $\mathcal{V}$ *that encode a subset of nodes in* V *with cardinality* k.

ACYLIC *From* (38) *we immediately see that* $\mathcal{A}$ *only contains arcs connecting neighboring layers and that the arcs always point towards the layer of greater cardinality.*

MULTIGRAPH $\mathcal{A}$ *is a multiset: multiple arcs connect the same pairs of nodes in* $\mathcal{G}$*. Parallel arcs might have different costs depending on their preimage edge. Additionally, for two parallel arcs* a, $b \in \mathcal{A}$*, there always holds* $a^{-1} \neq b^{-1}$.

Given the transition graph $\mathcal{G}$ and the arc costs γ, we set the source node of the new One-to-One MOSP instance to be node in $\mathcal{V}$ encoding the subset of nodes in V that contains s only, i.e., $\{s\} \in \mathcal{V}$. The target node for the new One-to-One MOSP instance is set to be the transition node encoding the whole set V of nodes in G, i.e., $V \in \mathcal{V}$ is the target node. Figure 28 includes an example of a MO-MST instance and its corresponding One-to-One MOSP instance $(\mathcal{G} = (\mathcal{V}, \mathcal{A}), \{s\}, V, d, \gamma)$. As noted in this section's introduction, this is the same One-to-One MOSP instance considered in (Santos et al., 2018).

13.2.1 Equivalent efficiency of trees in G and paths in $\mathcal{G}$ (cf. Santos et al., 2018, Section 3.1)

First we formalize when a path in $\mathcal{G}$ represents a tree in G.

Definition 13.4 (Path representations of a tree)**.** Consider a MO-MST instance (G, s, d, c) and the associated transition graph $\mathcal{G}$. For a node $U \in \mathcal{V}$ an $\{s\}$-U-path p in $\mathcal{G}$ is said to *represent* or *be a representation* of a spanning tree t of $G(U)$ if $t = \{a^{-1} \mid a \in p\}$.

The following lemma describes the mapping between trees in G and paths in $\mathcal{G}$ and their equivalent costs. Thereby it is important to note that a tree is represented by multiple paths but a path represents only one tree. The omitted proof corresponds to the proofs in (Santos et al., 2018, Proposition 3.1., Proposition 3.2.).

Lemma 13.2. *Consider a node subset* $U \subseteq V$ *in* G *containing* s*. A tree* t *in* $G(U)$ *rooted at* s *is represented in* $\mathcal{G}$ *by at least one* $\{s\}$*-*U*-path. For any such path* p *we* *have* $c(t) = \gamma(p)$*. Conversely, an* $\{s\}$*-*U*-path in* $\mathcal{G}$ *for some* $U \in \mathcal{V}$ *represents a tree* t *in* $G(U)$ *and* $\gamma(p) = c(t)$.

We can now conclude that a given MO-MST instance can be solved computing the efficient paths in a One-to-One MOSP instance on the induced transition graph. The statement is implicit in the exposition in (Santos et al., 2018, Section 3.1.). As we did in previous chapters if P is a set of paths in $\mathcal{G}$, we write $\gamma(P) := \{\gamma(p) \mid p \in P\} \subset \mathbb{R}^d_{\geqslant 0}$ to refer to the set of costs induced by the paths in P.

Theorem 13.2. *Consider a subset* $U \subseteq V$ *that contains* s*. A minimal complete set of efficient* $\{s\}$*-*U*-paths in* $\mathcal{G}$ *w.r.t.* γ *corresponds to a minimal complete set of efficient spanning trees of* $G(U)$ *w.r.t.* c.

Proof. Let P_U the set of $\{s\}$-U-paths in $\mathcal{G}$. By Lemma 13.2 every such path represents a spanning tree of $G(U)$ and all spanning trees of $G(U)$ are represented by at least one of these paths. Moreover, we have $\gamma(P_U) = c(T_{G(U)})$ and as a consequence, $\gamma(P_U)^* = c(T_{G(U)})^*$ which finishes the proof. □

13.2.2 Size of the transition graph

From this section onward we deviate from the exposition in (Santos et al., 2018). We begin with the analysis of the size of the biggest possible transition graph. This is important for two reasons. First, it creates awareness of the importance of applying *pruning methods* that reduce the size of the transition graphs by deleting arcs (cf. Section 13.2.3-Section 13.2.5). Second, the size of the transition graph is needed to derive an asymptotic running time bound for our new MO-MST algorithm (cf. Theorem 13.4). We consider a d-dimensional MO-MST instance (G, s, d, c) in which G is a complete graph with n nodes. Additionally, we again assume that all spanning trees are rooted at a node $s \in V$ and build the equivalent One-to-One MOSP instance $(\mathcal{G}, \{s\}, V, d, \gamma)$ as defined in Definition 13.3.

For a subset $U \subseteq V$ that contains s, let t be a spanning tree of $G(U)$ with depth one, i.e., a spanning tree in which every node other than the root node is a leaf node. Then, there are $(|U| - 1)!$ $\{s\}$-U-paths in $\mathcal{G}$. This number of representations of a tree in $\mathcal{G}$ is an upper bound because for trees with depth greater than one, the ordering of the edges along the root-to-leaf paths in the tree have to be respected along the $\{s\}$-U-paths in $\mathcal{G}$. If the spanning tree t is a path, then the representation of t in $\mathcal{G}$ by a $\{s\}$-U-path is unique.

In our setting, having fixed a node as the root node of all spanning trees, it is easy to see that for any $k \in \{1, \ldots, n\}$ the kth layer (cf. Remark 13.1) of nodes in $\mathcal{V}$ is build out of $\binom{n-1}{k-1}$ nodes. Thus, we obtain

$$|\mathcal{V}| = \sum_{k=1}^{n} \binom{n-1}{k-1} = 2^{n-1}. \tag{39}$$

The overall number of arcs between layer k and layer $k+1$ for any $k \in \{1, \ldots, n\}$ is

$$\binom{n-1}{k-1}(n-k)k \tag{40}$$

where the first multiplier is the number of transition nodes in the kth layer, the second multiplier is the number of missing nodes from G in any subset of nodes of G with cardinality k, and the last multiplier accounts for the number of parallel arcs. Thus, all in all, the number of arcs in $\mathcal{G}$ is

$$|\mathcal{A}| = \sum_{k=1}^{n} \binom{n-1}{k-1}(n-k)k = 2^{n-3}(n-1)n. \tag{41}$$

Clearly, the size of the transition graph $\mathcal{G}$ is an issue to address in the development of efficient Dynamic Programming algorithms for MO-MST problems. In the following sections, we discuss different techniques to reduce the size of $\mathcal{G}$. In Section 13.2.3, we discuss a polynomial running time approach that acts on the input graph G directly recognizing edges that can be deleted from the graph or edges that can be included in every efficient spanning tree. In Section 13.2.4 and Section 13.2.5, we discuss conditions to reduce the size of $\mathcal{A}$. These conditions are cut-dependent and are applied during the construction of the sets $\delta^+(U)$ for $U \in \mathcal{V}$. This stands in contrast to the pruning conditions used in the state of the art Dynamic Programming MO-MST algorithm introduced in (Santos et al., 2018). The so called *Built Network* algorithm ensures that every considered path represents a different tree. To do so, it is equipped with path-dependent pruning conditions that do not reduce the number of arcs in $\mathcal{A}$.

13.2.3 Polynomial time graph reduction during preprocessing

The manipulation of the input graph G described in this section is explained in (Sourd & Spanjaard, 2008). We refer the reader to the original paper to understand the details and correctness proofs. An edge $e \in E$ can be irrelevant for the MO-MST search if e is not contained in any efficient spanning tree of G or if for every nondominated cost vector, there exists a spanning tree with this cost vector that contains e.

Definition 13.5 (Red and Blue Arcs. cf. (Sourd & Spanjaard, 2008), Section 3.2)**.** Consider an instance (G, s, d, c) of the MO-MST problem and let T_G be the set of spanning trees of G.

BLUE EDGE An edge $e \in E$ is *blue* if for every nondominated cost vector in $c(T_G)^*$ there is an efficient tree $t \in T_G$ that contains e.

RED EDGE An edge $e \in E$ is *red* if there exists a minimal complete set T_G^* of efficient spanning trees of G for which e is not contained in any tree in the set.

Red edges can be deleted from G without impacting the final set of efficient spanning trees. In case $e = [u, v]$ is a blue edge, u and v can be contracted to build a new node w with $\delta(w) = (\delta(u) \cup \delta(v)) \setminus \{[u, v]\}$. Once w is built, u and v can be deleted from G.

Interestingly, using Sourd and Spanjaard (2008, Algorithm 3) the conditions to find blue and red edges can be checked in $\mathcal{O}\left(m^2\right)$ time. From now on, we only consider input graphs to our MO-MST instances that do not contain blue or red edges. By doing so, we assume that the described preprocessing phase is conducted before starting any actual MO-MST algorithm.

13.2.4 Pruning parallel transition arcs

The red and the blue conditions for edges in E, reduce the size of the input graph G. In other words, a red or a blue edge in G has no arc copies (cf. Definition 13.3) in $\mathcal{G}$. Using the condition that we derive in this section, we are only able to delete a subset of an edge's arc copies. In this sense, the condition from Lemma 13.4 is more *local* than the blue/red edge conditions.

The condition is motivated by an almost trivial condition that holds for every One-to-One MOSP instance.

Lemma 13.3. *Let* $(\mathcal{G}, \{s\}, V, d, \gamma)$ *be a One-to-One MOSP instance. If* $\mathcal{G}$ *contains two parallel arcs* a, a' *s.t.* $\gamma(a) \leqslant \gamma(a')$ *and* $\gamma(a) \neq \gamma(a')$ *then* a' *is not contained in any efficient* $\{s\}$-V*-path.*

The condition becomes more powerful if we study the condition's meaning regarding the preimage edges of the involved arcs.

Lemma 13.4. *Let* U *be a node set in* G *containing* s *and* $[u, w]$, $[u', w]$ *edges in* $\delta(U)$ *with* $u, u' \in U$ *and* $w \in V \backslash U$. *If* $[u, w]$ *dominates* $[u', w]$, $[u', w]$ *is not contained in any efficient spanning tree of* G *that contains an efficient spanning tree of* $G(U)$ *as a subtree.*

Proof. Let t_U be an efficient spanning tree of $G(U)$ contained in an efficient spanning tree t of G. If additionally t contains an edge $e = [u, w] \in \delta(U)$ with $u \in U$ and $w \notin U$, e must not be dominated by any edge $e' = [u', w] \in \delta(U)$ because otherwise $t_U \circ e$ would be dominated by $t_U \circ e'$. Thus, we could replace $[u, w]$ by $[u, w']$ in t and obtain a spanning tree of G that dominates t. This would contradict the assumption on t's efficiency. $\square$

To explain the last condition in terms of the transition graph, consider a transition node $U \in \mathcal{V}$. We call every transition node $W \in \mathcal{V}$ a *successor node of* U if there exists a path from U to W in $\mathcal{G}$. Since $\mathcal{G}$ is acyclic (cf. Remark 13.1), this notion is well defined. Assume there exists a dominance relation between parallel outgoing arcs of U, i.e., between two arcs $a, a' \in \delta^+(U)$ whose head node is a transition node $W = U \cup \{w\}$ for some $w \in G$. Then if a' is dominated by a, no copies of the preimage edge a'^{-1} need to be considered when building the sets $\delta^+(W')$ for any transition node W' that is a successor node of U.

13.2.5 Pruning dominated outgoing arcs

Our last condition (Lemma 13.5) to remove arcs from the transition graph is the most local one: it allows to remove single arcs $a \in \mathcal{A}$. From this condition, we cannot gain additional information from a's preimage edges to delete multiple copies of a^{-1} from $\mathcal{G}$.

The condition is a reformulation of a known necessary condition for the efficiency of trees already stated in e.g. (Hamacher & Ruhe, 1994; Corley, 1985; Chen et al., 2007). We reformulate it to emphasize that the condition holds for all efficient spanning trees of a given (sub)graph. For us, this means that we can use the condition to delete transition arcs without the need to check the condition for single trees.

Lemma 13.5. *Consider a MO-MST instance* (G, s, d, c). *Let* t *be an efficient spanning tree of* G. *For every edge* $e \in t$ *there is a cut* $\delta(U)$ *in* G *and a minimal complete set* $\delta(U)^*$ *of efficient cut edges s.t.* $e \in \delta(U)^*$.

Proof. Fix an edge $e \in t$ and consider the two disjoint trees t_U and t_W obtained after removing e from t. We assume that t_U is the tree containing the root node s and set $U = V(t_U)$ to be its node set. We have $e \in \delta(U)$ and any other cut edge $e' \in \delta(U)$ connects t_U and t_W hence building a new spanning tree t' of G. If a cut edge e' in $\delta(U)$ dominates e we get

$$c(t) = c(t_U) + c(e) + c(t_W) \leqslant c(t_U) + c(e') + c(t_W) = c(t').$$

For at least one index $i \in \{1, \dots, d\}$ we have $c_i(t) < c_i(t')$ implying t's dominance and thus contradicting the statement's assumption. □

Thus, when building the transition graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ we do not need to include arc copies in $\delta^+(U)$ of edges that are dominated in the cut $\delta(U)$ of G. Assume the edge $e \in \delta(U)$ is dominated but contained in an efficient spanning tree t of G. Lemma 13.5 guarantees that there is a node set $W \subset V$ for which e is an efficient cut edge in $\delta(W)$. Then, e is contained in a set of outgoing arcs $\delta^+(W)^*$ of the transition node $W \in \mathcal{V}$. All in all, we obtain the following result.

Theorem 13.3. *Consider a MO-MST instance* (G, s, dc) *without blue or red arcs in* G. *Any minimal complete set of efficient* $\{s\}$-V*-paths in the* Pruned Transition Graph $\mathcal{G}^* = (\mathcal{V}, \mathcal{A}^*)$ *with* $\mathcal{A}^* := \cup_{U \in \mathcal{V}} \delta^+(U)^*$ *corresponds to a minimal complete set of efficient* $\{s\}$-V*-paths in* $\mathcal{G}$.

Remark 13.2 (Incoming Arcs)**.** *From now on, we refer to the sets of incoming arcs of a transition node* $W \in \mathcal{V}$ *by* $\delta^-(W)^*$ *but the notation can be misleading. We use it to emphasize that we only consider a transition graph with the set* $\mathcal{A}^*$ *of arcs. However, even if the sets* $\delta^+(U)^*$ *do not contain dominated arcs for any* $U \in \mathcal{V}$, *the set* $\delta^-(W)^* := \{(U, W) \mid (U, W) \in \mathcal{A}^*\}$ *can contain arcs that dominate each other.*

13.3 NEW DYNAMIC PROGRAMMING ALGORITHM

Remark 13.3. *In this chapter, we introduce our IG-MDA and discuss our version of the BN algorithm. Both are label-setting One-to-One MOSP algorithms. Thus, our implementations use speedup techniques such as dimensionality reduction or target pruning. Since these techniques are described in detail in Section 3.4 and in Chapter 9, we give a higher level description of the algorithms in this chapter. This makes the exposition less cluttered and allows us to focus on the peculiarities of the considered One-to-One MOSP instances.*

In the last section, we described the translation from a given MO-MST instance (G, s, d, c) with all trees rooted at a node $s \in V$ to the corresponding equivalent One-to-One MOSP instance $\mathcal{I}_{SP} = (\mathcal{G}^*, \{s\}, V, d, \gamma)$. In this section, we explain how to use a slightly modified version of the tmda for One-to-One MOSP instances like $\mathcal{I}_{SP}$. We call the resulting algorithm the *Implicit Graph Multiobjective Dijkstra Algorithm* (IG-MDA). The main innovation in the IG-MDA targets possible issues with the size of $\mathcal{G}^*$. It does so creating the graph implicitly as needed when paths are expanded. The downside of this approach is that, since the graph is not known a priori, we cannot compute a heuristic to build the reduced costs $\bar{\gamma}$ and guide the search towards the target node. However, it shall be noted right away that we tested the resulting heuristic in early stages of our development of our MO-MST algorithm and found out that building $\mathcal{G}^*$ explicitly is, in most cases, a prohibitive task in terms of running time and memory consumption.

Algorithm 12 is the pseudocode of the IG-MDA. In our pseudocode, we assume the existence of a container T^* that stores the lists $T^*_{G(U)}$ for every $U \in \mathcal{V}$. We proceed with a high level description of the algorithm in the remainder of the section. Most parts of the algorithm coincide with the ones explained and proven in Chapter 9. Readers of this thesis interested in in-depth discussions and proofs about the IG-MDA are referred to Chapter 9 and the tmda introduced therein. The implicit handling of $\mathcal{G}^*$ in the IG-MDA is discussed in Section 13.3.1.

Algorithm 12: Implicit Graph Multiobjective Dijkstra Algorithm (IG-MDA)

```
Input  : d-dimensional One-to-One MOSP instance $\mathcal{I}_{SP} = (\mathcal{G}^*, \{s\}, V, d, \gamma)$.
Output: Minimal complete set $T^*_{G(V)}$ of $\{s\}$-$V$-paths in $\mathcal{G}$ w.r.t. $\gamma$.
1  Prio. queue of paths $Q \leftarrow \emptyset$;          // Lex. non-decreasing order w.r.t. $\gamma$.
2  Transition graph $\mathcal{G}$ initialized only containing the transition node $\{s\}$;
3  Trivial $\{s\}$-$\{s\}$-path $p_{init} \leftarrow ()$;
4  $Q \leftarrow Q.\text{insert}(p_{init})$;
5  while $Q \neq \emptyset$ do
6      $p \leftarrow Q.\text{extractMin}()$ ;
7      $U \leftarrow$ last transition node of path $p$ ;          // If $p = p_{init}$, $U \leftarrow \{s\}$.
8      success $\leftarrow$ False;
9      if $U \neq V$ then $(Q, \text{success}) \leftarrow$ ig-propagate$(p, Q, T^*, \mathcal{NQP})$ ;
10     if $U == V$ or success == True then $T^*_{G(U)}.\text{append}(p)$;
11     New queue path $p'$ for $U \leftarrow$ Solve (42) according to $U$ and the existing
         $\mathcal{NQP}_a$ lists for $a \in \delta^-(U)^*$ ;
12     if $p' \neq$ NULL then $Q.\text{insert}(p')$;
13 return $T^*_{G(V)}$;
```

In every iteration of the IG-MDA, a lex. minimal explored path is extracted from the queue Q (Line 6). Assume p is an extracted $\{s\}$-U-path for some node $U \in \mathcal{V}$. Since p is an explored path, it is not dominated by or equivalent to any path in $T^*_{G(U)}$ by definition. Since for any $U \in \mathcal{V}$ the explored path in Q is lex. minimal compared to other existing explored $\{s\}$-U-paths, p is guaranteed to be an efficient $\{s\}$-U-path in $\mathcal{G}$ w.r.t. γ (cf. Definition 3.5). Thus, when extracted from Q, the IG-MDA builds the expansions of p along the outgoing arcs in $\delta^+(U)^*$ and decides if p needs to be stored in $T^*_{G(U)}$.

Algorithm 13: ig-propagate.

```
Input  : $\{s\}$-$U$-path $p$, priority queue $Q$, permanent paths $T^*$, lists of
         explored paths $\mathcal{NQP}$.
Output : Updated priority queue $Q$ and a boolean flag telling if the
         propagation of $p$ was successful along at least arc in $\delta^+(U)^*$.
1  Flag success ← False;
2  if $\delta^+(U)^*$ not initialized then build $\delta^+(U)^*$ as described in Section 13.3.1 ;
3  for $W \in \delta^+(U)^*$ do
4      $q \leftarrow p \circ (U, W)$;
5      if $\gamma(T^*_{G(W)}) \preceq_D \gamma(q)$ then continue;
6      success ← True;
7      if Q does not contain an $\{s\}$-$W$-path then
8          Q.insert (q) ;
9      else
10         $q' \leftarrow$ $\{s\}$-$W$-path in Q;
11         if $\gamma(q) \prec_{lex} \gamma(q')$ then
12             Q.decreaseKey(W, q) ;
13             if not $\gamma(q) \leqslant \gamma(q')$ then
14                 $(U', W) \leftarrow$ last arc in path $q'$;
15                 Insert $q'$ at the beginning of $\mathcal{NQP}_{(U',W)}$;
16         else
17             if not $\gamma(q') \leqslant \gamma(q)$ then Insert q at the end of $\mathcal{NQP}_{(U,W)}$ ;
18 return (Q, success);
```

The decision is made in the subroutine ig-propagate called in Line 9 of the IG-MDA. The subroutine works like the propagate subroutine from the tmda. It uses $\mathcal{NQP}$ lists associated with the arcs in $\mathcal{G}^*$ to store the explored paths that are not queue paths. If the expansion of an efficient $\{s\}$-U-path p along the arcs in $\delta^+(U)^*$ does only produce dominated or equivalent paths, p is not stored in $T^*_{G(U)}$.

Besides the propagation of p as described in the last paragraph, every iteration of the IG-MDA needs to search for a new explored $\{s\}$-U-path p' to be stored in the priority queue Q. The new path has to be a queue path for U according to Definition 9.3. The candidate $\{s\}$-U-paths are stored in the $\mathcal{NQP}_a$ lists for $a \in \delta^-(U)^*$ and thus, p' solves the minimization problem

$$\arg\operatorname{lex}\min \left\{ \gamma(p) \middle| p \in \mathcal{NQP}_a,\ a \in \delta^-(U)^*, \text{ and not } \gamma(T^*_{G(U)}) \preceq_D \gamma(p) \right\}. \tag{42}$$

The minimization and the addition of p' to Q in case such a new explored path is found happen in Line 11 and Line 12 of Algorithm 12. The routine nextQueuePath* in the tmda is used to solve (42).

Further details on how the original tmda proceeds, its correctness, and further speedup techniques in the implementation are described in Chapter 9.

13.3.1 Implicit Handling of the Transition Graph

In this section we explain how the transition graph $\mathcal{G}$ is managed implicitly in the IG-MDA. The addition to a node U to the initially empty set $\mathcal{V}$ of transition nodes always entails the initialization of the list $T^*_{G(U)}$ since we assume that the algorithm will store some permanent and thus efficient $\{s\}$-$\mathcal{U}$-paths. No arcs are added during this node initialization. At the beginning of the algorithm, the IG-MDA only adds the node $\{s\}$ to $\mathcal{G}$ (Line 2).

When an $\{s\}$-$\mathcal{U}$-path p is extracted from the priority queue, expansions of p along outgoing arcs of $\mathcal{U}$ are built in ig-propagate. To this aim, the set $\delta^+(\mathcal{U})^*$ needs to be constructed if it does not yet exist (Line 2 of ig-propagate). The construction happens using (38) and the arc removal techniques discussed in Lemma 13.4 and Lemma 13.5. The creation of an arc $(\mathcal{U}, \mathcal{W})$ in this set requires

- the addition of $(\mathcal{U}, \mathcal{W})$ to the set $\delta^+(\mathcal{U})^*$,
- the addition of $\mathcal{W}$ to $\mathcal{V}$ as explained in the last paragraph if it does not yet exist,
- the addition of $(\mathcal{U}, \mathcal{W})$ to $\delta^-(\mathcal{W})^*$,
- and the initialization of the list $\mathcal{NQP}_{(\mathcal{U},\mathcal{W})}$.

In case the set $\delta^+(\mathcal{U})^*$ already exists when p is extracted from the queue, we are sure that the lists $\mathcal{NQP}_{(\mathcal{U},\mathcal{W})}$ that are possibly needed in ig-propagate are already initialized. Moreover, the new explored paths $q = p \circ (\mathcal{U}, \mathcal{W})$ obtained from p's expansions can be added to $T^*_{G(\mathcal{W})}$ if needed later.

13.3.2 Running Time

Using the running time bound for the mda and the tmda derived in Theorem 4.4, the number of nodes (39), and the number of arcs (41) in $\mathcal{G}$, we obtain the following result.

Theorem 13.4. *Consider an MO-MST instance* (G, s, d, c) *and the corresponding One-to-One MOSP instance* $\mathcal{I}_{SP} = (\mathcal{G}^*, \{s\}, V, d, \gamma)$*. Using the IG-MDA to solve* $\mathcal{I}_{SP}$*, setting* $N_{\max} := \max_{\mathcal{U}\in\mathcal{V}} |T^*_{G(\mathcal{U})}|$*, and applying Theorem 13.2,* (G, s, d, c) *can be solved in*

$$\mathcal{O}\left(dN_{\max}\Big(|\mathcal{V}|\log|\mathcal{V}| + |\mathcal{A}|N_{\max}\Big)\right).$$

Note that the running time is not output sensitive since the the transition graph can contain polynomially many nodes and arcs w.r.t. the size of the input graph G.

13.4 BUILD NETWORK ALGORITHM

In this section, we briefly describe the *Build Network* (BN) algorithm from (Santos et al., 2018), using our terminology to compare it with the IG-MDA in what follows.

Again, we start with a given a MO-MST instance (G, s, d, c) without red or blue edges. For the description of the algorithm we again need the One-to-One MOSP instance $\mathcal{I}_{SP} = (\mathcal{G}, \{s\}, V, d, \gamma)$. Note that the original transition graph $\mathcal{G}$ is used instead of $\mathcal{G}^*$. Then, the BN algorithm looks for efficient $\{s\}$-V-paths w.r.t. γ in $\mathcal{G}$. Explored paths are stored in a priority queue Q that is sorted lexicographically. merge operations are conducted in every iteration to delete dominated explored paths. In our implementation of the BN algorithm we manage explored paths as in the NAMOA$^*_{dr}$-lazy algorithm.

For every new path p extracted from the algorithm's queue, the algorithm analyzes the ordering in which arcs were added to p to determine the allowed expansions of p. The conditions ensure that no two paths considered by the BN algorithm encode the same tree in G. This is remarkable given that a tree with k nodes can have up to $(k-1)!$ representations in $\mathcal{G}$ as noted in Section 13.2.2. These unique representations of trees in $\mathcal{G}$ are called *minimal paths* (cf. Santos et al., 2018, Definition 3.5.)).

Informally, minimal paths can be defined as follows. The definition is recursive and assumes that nodes in G are labeled with unique ids from 1 to n. A path with only one arc is always minimal. Let p be a minimal $\{s\}$-U-path in $\mathcal{G}$ for some $U \in \mathcal{V}$ and let t be the spanning tree of $G(U)$ induced by p. The preimage edges of the arcs in p, in their order of appearance along p, define a permutation π_p that maps the ids of the nodes in U to their order of inclusion into t. Now, let $[u, w]$ be the preimage edge of the last arc in p. There are two possibilities for the expansions of p to be minimal paths. If edges $[u, w'] \in \delta(u)$ s.t. $w' \notin U$ exist, then the paths obtained after expanding p along the copies of the arcs $[u, w']$ in $\delta^+(U)$ s.t. the id of w' is greater than the id of w are minimal paths. In the second option, we assume that $\pi(u) = k$. It is guaranteed that $k \leqslant |U|$. Then, p must be expanded along the copy of an edge $[u', w']$ in $\delta^+(U)$ s.t. $\pi(u') > k$. We assume w.l.o.g that $u' \in U$ and $w' \notin U$.

Example 13.1 (Minimal Paths)**.** Consider the complete graph K_4 with four nodes. The path p with the preimage edges $([1, 3], [3, 4])$ is a minimal path. The corresponding tree spans the subgraph with nodes $U = \{1, 3, 4\}$. A spanning tree of the K_4 is obtained by adding any of the edges $[1, 2]$, $[3, 2]$, $[4, 2]$ to the tree. I.e., p must be expanded along the copies of one of these edges in $\mathcal{G}$. However, expanding along a copy of $[3, 2]$ does not lead to a minimal path. The reason is that the preimage of the last arc in p is $[u, w] = [3, 4]$ and $[3, 2]$ has the same node in U, namely node u with id 3 and a node in $V \setminus U$, namely node w' with id 2, that has a smaller id than 4. Additionally, the expansion of p along the copy of the edge $[u', w'] = [1, 2]$ does also not yield a minimal path because $\pi(u) = \pi(3) = 2$ and $\pi(u') = \pi(1) = 1 > 2$ which contradicts the second condition for minimal paths. The expansion of p along the copy of the edge $[u', w'] = [4, 2]$ does indeed yield a minimal path: we have $\pi(4) = 3 > 2 = \pi(u)$ and the second condition for minimal paths holds.

The pseudocode of our version of the BN algorithm is in Algorithm 14. We combine the minimal path definition from (Santos et al., 2018) with a *lazy queue management approach* for explored paths as in the NAMOA$^*_{dr}$-lazy algorithm to notably enhance the running time of the BN algorithm in our experiments. Assume the minimal $\{s\}$-W-path q in $\mathcal{G}$ is considered by the algorithm. It is immediately discarded (Line 14) if it is dominated by or equivalent to any path in $T^*_{G(W)}$. Otherwise, it is inserted into the algorithm's priority queue Q without further checks (Line 15). In Algorithm 14 we repeat the dominance or equivalence check $\gamma(T^*_{G(W)}) \preceq_D \gamma(q)$ after q's extraction from Q (Line 8). If q passes the check, it is expanded and possible made permanent.

Algorithm 14: Built Network (BN) algorithm.

```
Input  : d-dimensional One-to-One MOSP instance $\mathcal{I}_{SP} = (\mathcal{G}, \{s\}, V, d, \gamma)$.
Output: Minimal complete set $T^*_{G(V)}$ of efficient $\{s\}$-$V$-paths in $\mathcal{G}$ w.r.t. $\gamma$.
1  Priority queue of paths $Q \leftarrow \emptyset$;  // Sorted lexicographically according to $\gamma$.
2  Transition graph $\mathcal{G}$ initialized only containing the transition node $\{s\}$;
3  Trivial $\{s\}$-$\{s\}$-path $p_{init} \leftarrow ()$;
4  $Q \leftarrow Q.insert(p_{init})$;
5  while $Q \neq \emptyset$ do
6      $p \leftarrow Q.extractMin()$ ;
7      $U \leftarrow$ last transition node of path $p$ ;        // If $p = p_{init}$, $U \leftarrow \{s\}$.
8      if $\gamma(T^*_{G(U)}) \preceq_D \gamma(p)$ then continue;
9      Boolean flag success $\leftarrow$ False;
10     if $\delta^+(U)$ not initialized then build $\delta^+(U)$ as described in Section 13.4.1 ;
11     $\mathcal{MIN}_p \leftarrow$ Minimal paths in $\{p \circ a \mid a \in \delta^+(U)\}$ ;
12     for $q \in \mathcal{MIN}(p)$ do
13         Assume $q$ is spanning tree of $G(W)$ for $W \subseteq V$;
14         if not $\gamma(T^*_{G(W)}) \preceq_D \gamma(q)$ then
15             $Q.insert(q)$;
16             success $\leftarrow$ True;
17     if success == True then $\mathcal{P}_U.append(p)$;
18 return $T^*_{G(V)}$;
```

The correctness of our version of the BN algorithm follows from the fact that considering minimal paths in $\mathcal{G}$ suffices to find a minimal complete set of efficient paths that represents a minimal complete set of efficient spanning trees in G (Santos et al., 2018, Proposition 3.7.) and from the correctness of the NAMOA$^*_{dr}$-lazy algorithm. All in all, we have designed a new algorithm combining the elegant pruning rule (minimal paths) that made the original BN algorithm state of the art with recent advances used to improve the handling of explored paths in label-setting MOSP algorithms.

13.4.1 Implicit Handling of the Transition Graph

Handling the transition graph $\mathcal{G}$ implicitly is easier for the BN algorithm than for the IG-MDA. It works as described in Section 13.3.1 with the following two differences. First, neither lists of incoming arcs nor $\mathcal{NQP}$ lists have to be maintained. Second, the lists of $\delta^+(U)$ of outgoing arcs for a transition node U are built in Line 10 as described in (38) without any arc removal conditions. For every $\{s\}$-U-path p extracted from Q the relevant outgoing arcs needed to build minimal path expansions of p are chosen in a path-dependent way in Line 11. We suppose that the authors of Santos et al. (2018) also generate the transition graph $\mathcal{G}$ implicitly as described in this section to restrict the memory consumption of $\mathcal{G}$ as far as possible.

13.4.2 Comparison to the IG-MDA

Finally, before moving on to the experiments in this chapter, we briefly summarize the differences between the IG-MDA and our implementation of the BN algorithm. All in all, there are two main differences between the IG-MDA and our new BN algorithm:

EXPLORED PATHS The IG-MDA restricts the size of the queue and stores other explored paths in $\mathcal{NQP}$ lists. It requires to search for a new queue path in every iteration but does not require a dominance or equivalence check after a path's extraction from the queue. Our new BN algorithm stores multiple explored paths per node in the queue but requires dominance or equivalence checks after every path's extraction from the queue. This difference is already present in the comparison of the tmda and the NAMOA$^*_{\text{dr}}$-lazy algorithm in Chapter 9.

PRUNING CONDITIONS The IG-MDA uses the cost-dependent conditions discussed in Lemma 13.4 and Lemma 13.5 to reduce the number of arcs in the transition graph in a path-independent way. The BN algorithm in its original version as well as in our implementation uses path-dependent pruning techniques (minimal paths, cf. Santos et al., 2018, Definition 3.5.) forcing the inclusion of nodes in the represented spanning trees to follow fixed rules.

13.5 EXPERIMENTS

All codes, results, and evaluation scripts used to generate the contents in this section are publicly available in (Maristany de las Casas, 2023c).

13.5.1 Implementation Details

As noted in the introduction, when choosing a benchmark algorithm for our experiments in this chapter, we benefit from the extensive survey on MO-MST algorithms by Fernandes et al. (2020). For more than two objectives, our use case, the best results are obtained using the BN algorithm

The deletion of red and blue edges from the original input graphs as explained in Section 13.2.3 is done in a preprocessing stage and both algorithms work on the resulting graphs without red or blue edges. In (Fernandes et al., 2020, Table 13) the authors list the number of red and blue edges in many MO-MST instances that we use in our experiments. In our code, we do no further manipulation of the input data.

Both algorithms use the dimensionality reduction technique discussed in Section 3.5. Note that in (Santos et al., 2018) the authors achieved their fastest running times using the *sum of cost components* as the sorting criterion for explored paths. Also in our experiments this criterion yields better running times than lex. sorting without using dimensionality reduction. However, adding the dimensionality reduction technique to the BN algorithm when using lex. sorting, the overall running time is clearly better than using the sum of cost components as sorting criteria. Therefore, we report the results obtained from our BN implementation using lex. sorting and the dimensionality reduction technique.

Further implementation details such as the representation of paths using labels to be able to store a path using $\mathcal{O}(1)$ memory (cf. Section 3.3) and the usage of a memory pool to avoid memory fragmentation can be read our source code (Maristany de las Casas, 2023c) directly. The running time of our implementation of the BN algorithm is dominated by the computation of minimal paths. Thus, it is worth noting that for every explored path in the BN algorithm, we store a vector indicating the order in which nodes of G are added to the corresponding tree. We then use this vector in Line 11 of Algorithm 14 to recognize relevant outgoing arcs faster in every iteration.

13.5.2 Instance Description

The authors from (Santos et al., 2018) kindly gave us access to their instances from which we use the 3- and 4-dimensional MO-MST instances for our experiments. The instances are called *SPACYC* instances. For each number of criteria, graphs with 5 to 14 nodes are generated. For each fixed number of n nodes, graphs with $m = ni$ edges for $i \in \{5, 10, 15, 20\}$ are generated. For each arc, the costs are generated randomly using the *SPACYC* generator from (Knowles & Corne, 2001). Thereby, the cost criteria are not correlated and the costs are chosen from the interval $[0, 100]$.

We also got access to the instances used in Fernandes et al. (2020). In this chapter, we use their 3- and 4-dimensional MO-MST instances. For each dimension, we consider grid graphs and complete graphs with varying number of nodes (see (Fernandes et al., 2020; Maristany de las Casas, 2023c) for details). Additionally, instances were generated with anticorrelated and correlated edge costs. For a fixed problem dimension (3 or 4 cost criteria), a fixed number of nodes, and an edge costs type there are 30 different instances. Thus, for example, we consider 30 complete graphs with 12 nodes and anticorrelated edge costs functions with 3 cost criteria. On all instances, the edge costs are in the interval $[0, 100]$.

13.5.3 Computational Environment

All computations were run on a machine with an Intel Xeon-Gold 6246 @ 3.30GHz processor. The source code, written in C++, was compiled using g++ v.7.5.0 using the compiler flag $-$O3. Both algorithms were granted 2h of time and 30GB of memory to solve each instance.

Indexing transition nodes depending on the node set from the original graph G that they represent is important in both algorithms. To achieve this, we use the dynamic bitset class from the boost library (Boost, 2022) and for every bitset of length $n-1$ (the node s is fixed) we compute the corresponding decimal representation to obtain an index. By doing so, we restrict ourselves to graphs with at most 64 nodes in today's 64bit systems. However, as we will see in this section, the amount of efficient trees in every considered graph type requires more than the available 30GB of memory before reaching our constraint on the number of nodes. We did experiments

Nodes	10	11	12	13	14
BN 3d in (Santos et al., 2018)	0.25	1.00	5.42	22.11	86.33
CPU scaled	0.19	0.76	4.14	16.87	65.87
BN 3d as in Algorithm 14	0.04	0.10	0.54	2.19	11.25
BN 4d in (Santos et al., 2018)	5.58	28.40	188.14	712.96	3656.91
CPU scaled	4.26	21.67	143.55	543.99	2790.22
BN 4d as in Algorithm 14	0.76	6.34	69.76	351.73	1737.88

Table 16: Comparison of BN algorithm implementations in (Santos et al., 2018) and in (Maristany de las Casas, 2023c). Computations using 3 and 4-dimensional SPACYC based instances.

granting as much memory as we could to both algorithms but also time became a restrictive factor. The number of efficient spanning trees in our MO-MST instances grows extremely fast and the results reported in this section contain the biggest instances we could solve.

13.5.4 Efficiency of our BN implementation

We tried to determine whether our implementation of the BN algorithm from (Santos et al., 2018) is efficient. Since we do not have access to the original implementation from the authors, we extrapolated their computational results to the performance on our machine. Using (PassMark-Software, 2023) we found out that on single-threaded jobs the processor used in (Santos et al., 2018) delivers 76.3% of the performance of our processor. We scaled the average running times for the SPACYC based instances (defined in Section 13.5) in (Santos et al., 2018, Table 6) accordingly to obtain the expected running times of their code on our machine. In Table 16 we list the original running times from (Santos et al., 2018), the CPU-scaled running times we obtained, and the running times we obtained from our implementation of Algorithm 14. Since the BN implementation in (Santos et al., 2018) is coded in Java and ours is coded in C++, the running times are still not completely comparable. However, our running times seem to be good enough ($\times 6$ faster on the biggest 3d instances and $\times 1.6$ faster on the biggest 4d instances) to claim that our implementation of the BN algorithm is efficient and useful for the comparisons that follow in this paper. The speedup decreases notably with increasing edge costs' dimension because the impact of $\preceq_D$ checks on running time increases in higher dimensions. Most probably, both programming languages are similarly efficient in performing these checks. Note that the reported running time averages in Table 16 do not coincide with those reported in Table 17. The reason is that in (Santos et al., 2018) the authors always use arithmetic means and in this paper we use geometric means everywhere except in Table 16.

13.5.5 Results

In this section we finally report the results obtained from our comparison between the IG-MDA and the BN algorithm implemented in (Maristany de

$\lvert V\rvert$	$\lvert T^*_G\rvert$	BN			IG MDA			Speedup
		$\lvert\mathcal{V}\rvert$	Iterations	Time	$\lvert\mathcal{V}\rvert$	Iterations	Time	
3d edge costs								
10	398.14	477.71	17846.60	0.0318	312.88	7496.75	0.0060	5.27
11	487.21	774.66	36049.40	0.0695	526.56	14310.60	0.0122	5.70
12	815.86	1609.97	107572.10	0.3579	1179.25	41966.32	0.0421	8.50
13	1073.61	3111.80	252588.01	1.3572	1789.97	76472.73	0.0882	15.39
14	1747.29	6835.16	862549.99	8.4189	4288.41	259474.39	0.5023	16.76
4d edge costs								
9	1146.20	241.41	15941.44	0.0461	197.35	10460.27	0.0163	2.83
10	3014.38	471.18	66369.32	0.5246	413.88	43938.54	0.1431	3.66
11	4532.14	996.44	195159.07	2.7925	859.96	120389.30	0.5378	5.19
12	9060.48	2002.31	705053.83	30.8805	1705.53	409772.59	3.7650	8.20
13	13471.15	3734.22	1859534.46	190.0873	3196.62	1061076.64	17.2987	10.99
14	20735.06	7470.75	5051489.41	862.0660	6237.86	2474854.74	57.3346	15.04

Table 17: SPACYC based 3d and 4d instances. For every node cardinality $\lvert V\rvert$, 20 instances were considered. Both algorithms solved every instance. Numbers are geometric means.

las Casas, 2023c). Note for small input graphs, the obtained running times were below 0.01s for both algorithms. In the tables in this section, we do not report results for any group of graphs for which both algorithms solved the instances in less than 0.01s. However, in the results folder in (Maristany de las Casas, 2023c), all results can be accessed and we also included the scripts used to generate the full tables and plots from this section. Also, for every table in the upcoming subsections, we pick a scatter plot corresponding to the running times of both algorithms that lead to one representative line of the table. The remaining scatter plots, one per line, are also stored in the results folder in (Maristany de las Casas, 2023c).

Results from SPACYC based instances

In Table 17 we summarize the results obtained from the SPACYC based instances. Both algorithms solve every instance. While in 3d the BN algorithm is faster than the IG-MDA on instances with less than 8 nodes, this effect disappears in 4d, where the IG-MDA outperforms the BN algorithm consistently for all graph sizes. In both dimensions, the speedup grows with the input graph size and reaches $\times 16.76$ on the 3d instances and $\times 15.04$ on the 4d instances. Thereby, the IG-MDA performs better regarding the metrics *iterations per second* and *iterations per efficient spanning tree*. The second metric also implies that the IG-MDA solves the instances in less iterations since the number of efficient spanning trees depends on the instances and not on the algorithm. Figure 29 and Figure 31 show, for graphs with 10 to 14 nodes, the number of iterations that the IG-MDA and the BN algorithm performed per second. The IG-MDA outperforms the BN algorithm regarding this metric consistently. Particularly in Figure 31 we observe that both algorithms seem to converge. This is because the dominance checks ($\preceq_D$) become more complex as the number of efficient solutions in the sets $T^*_{G(u)}$ for transition nodes u increases. Since these sets are equal for both algorithms, both algorithms make the same effort to decide whether explored trees are dominated.

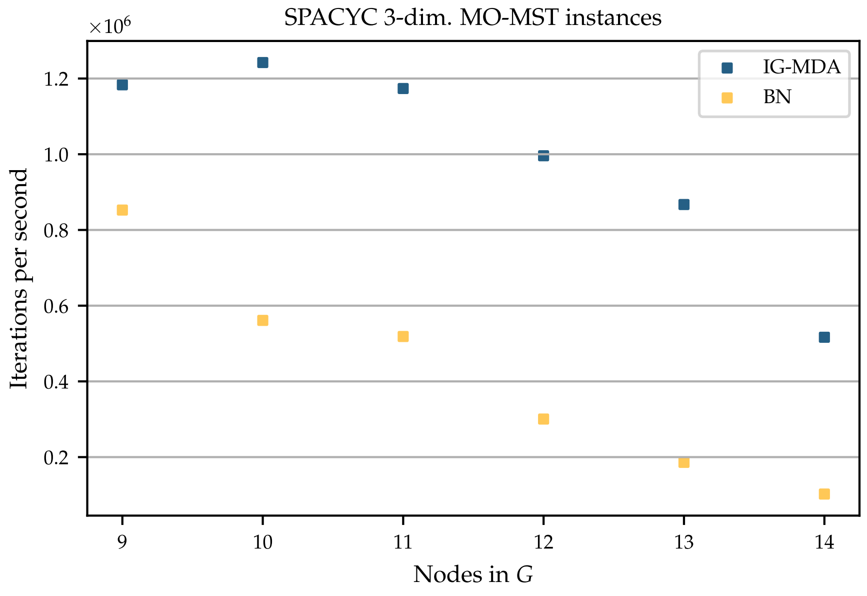


Figure 29: 3d SPACYC instances. Average iterations per second.

If the iterations per second of both algorithms converge but the speedup in favor of the IG-MDA increases with the graph size, the IG-MDA must do less iterations than the BN algorithm. Indeed, this can be already observed in Table 17. In Figure 30 and Figure 32 we plot the iterations per efficient spanning tree needed by both algorithms. This metric unveils that the effort made by the IG-MDA increases much slower as the graph size increases. Using *minimal paths*, the BN algorithm needs to decide upon the relevance of new explored paths using dominance checks only. Using cost-dependent arc pruning techniques as described in Section 13.2.4 and Section 13.2.5, the IG-MDA avoids the expansion of and $\preceq_D$ -checks for every efficient $\{s\}$-$\mathcal{U}$-path in $T^*_{G(\mathcal{U})}$ along a pruned arc $(\mathcal{U}, W)$. In particular, pruning all incoming arcs of a transition node W leads to less transition nodes in the implicit graphs of the IG-MDA algorithm (cf. $|\mathcal{V}|$ columns in Table 17).

Results from Fernandes et al. instances

We summarize the results in Table 18 and Table 19.

COMPLETE GRAPHS WITH ANTICORRELATED COSTS We report our results in Table 18. Both algorithms solve all instances with up to 12 nodes regardless of the edge costs' dimension. Thereby, the IG-MDA is ×6.03 faster on 3d instances and ×8.50 faster on 4d instances. Additionally, the IG-MDA solves all 3d anticorrelated instances with 15 nodes and all but one with 17 nodes. Regarding the 4d instances, it solves 22 instances with 15 nodes. This is an improvement w.r.t. to the biggest instances of this type solved in (Fernandes et al., 2020) (12 nodes in 3d and 10 nodes in 4d). Note that computations are aborted because the memory limit of 30GB is hit. Figure 33 and Figure 34 show the average running times of both algorithms for two relevant graph sizes. The same plots for every other considered graph size are in the result folder from (Maristany de las Casas, 2023c).

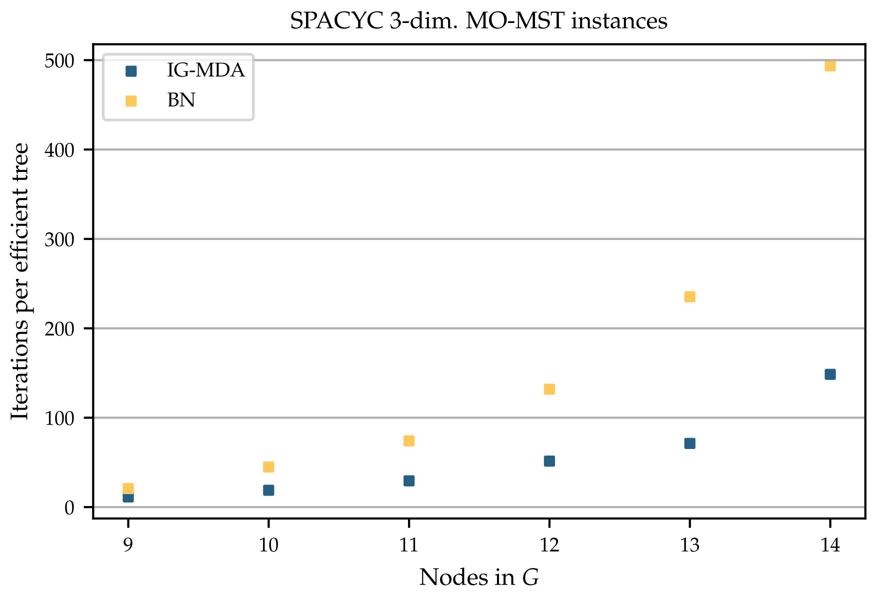


Figure 30: 3d SPACYC instances. Average iterations per efficient tree.

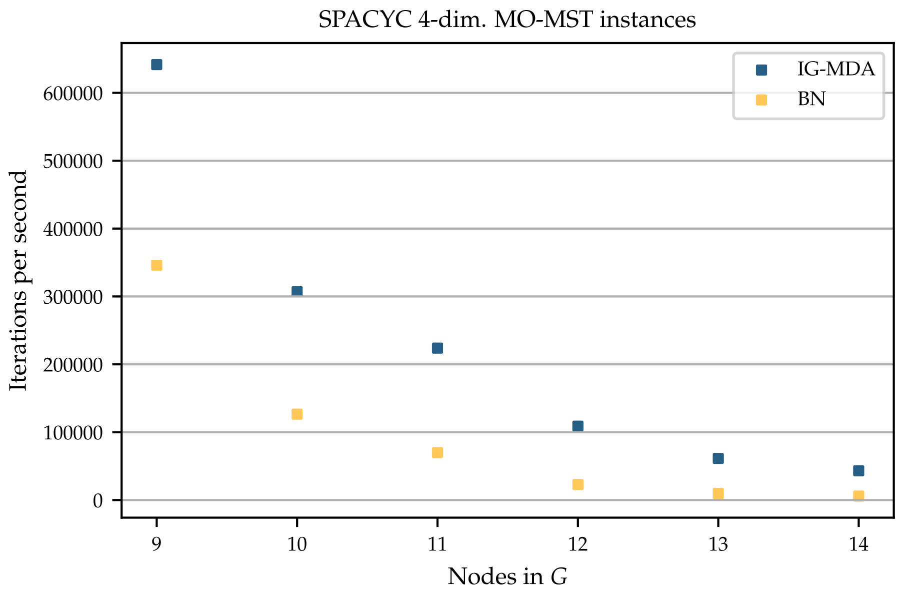


Figure 31: 4d SPACYC instances. Average iterations per second.

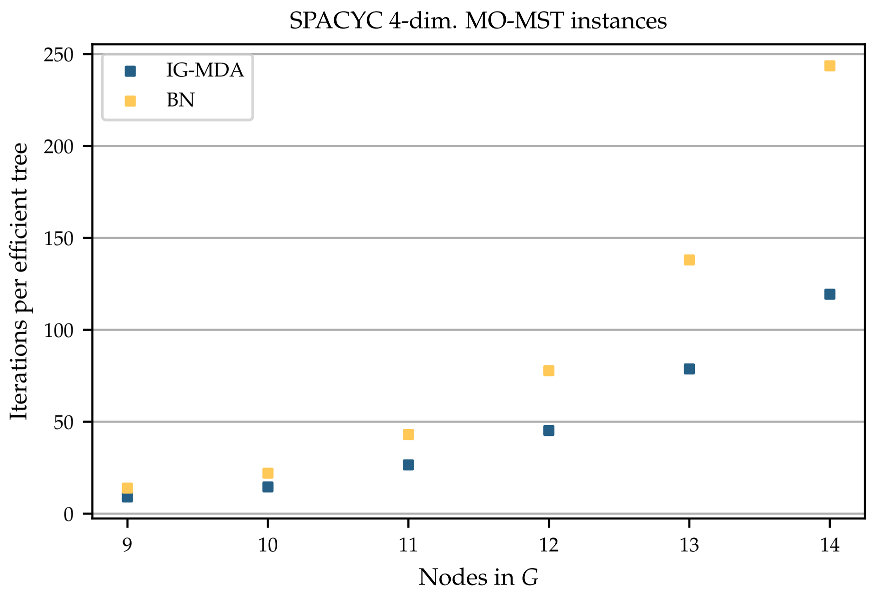


Figure 32: 4d SPACYC instances. Average iterations per efficient tree.

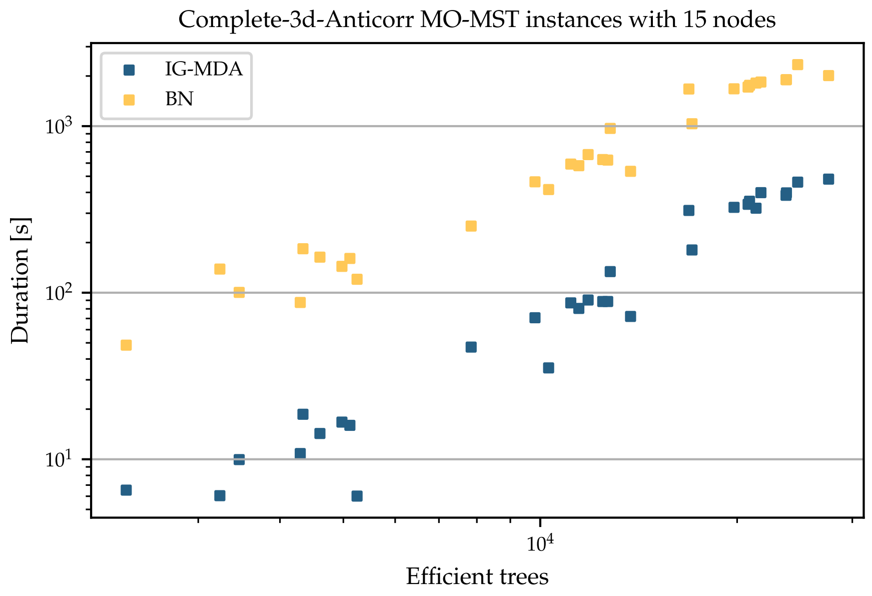


Figure 33: Running times on Complete-3d-Anticorr instances with 15 nodes.

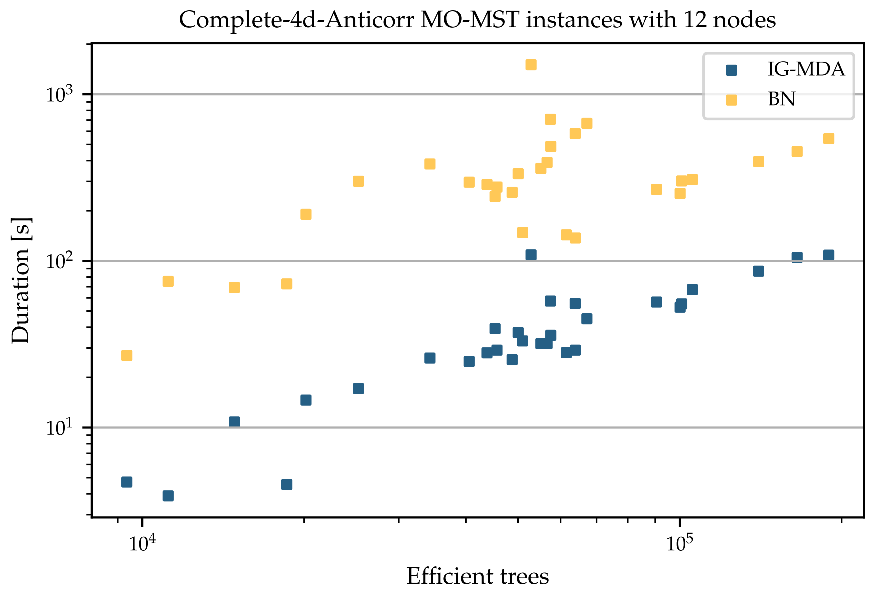


Figure 34: Running times on Complete-4d-Anticorr instances with 12 nodes.

GRID GRAPHS WITH ANTICORRELATED COSTS The results are summarized in Table 18. We observe the same trend as before regarding the speedups in favor of the IG-MDA. All instances with graphs with up to 18 nodes are solved in less than 0.01s by both algorithms. On 3d instances, both algorithms solve all instances with up to 24 nodes. On bigger graphs, the BN algorithm solves a subset of the instances solved by the IG-MDA. The IG-MDA starts failing to solve instances on input graphs with 33 nodes. There are seven instances with 38 nodes that are solved by both algorithms (the IG-MDA solves 11/30 such instances) and on these instances the IG-MDA is $\times 68.16$ faster (cf. Figure 35). Note that the biggest instances from this group solved in (Fernandes et al., 2020) were grids with 20 nodes. The reason why the BN algorithm manages to solve some instances with 38 nodes even though it could not solve any instance with 36 nodes and only two with 33 nodes is that the seven solved instances with 38 nodes contain multiple blue and red edges. Thus, the actual input graphs for both algorithms are smaller (for details, see (Maristany de las Casas, 2023c)). Regarding instances with 4d anticorrelated edge cost functions on grid graphs, both algorithms solve all instances with up to 24 nodes. The IG-MDA still solves all 30 instances with 27 nodes. From that size onward, the number of solved instances decreases because of the memory limit. The IG-MDA speedup on the instances with 24 nodes is $\times 2.99$ (see Figure 36). The relatively small number compared to other instance sets is because grids induce smaller transition graphs and, particularly on 4d instances, the running time is mostly determined by $\preceq_D$-checks that are equally time consuming for both algorithms.

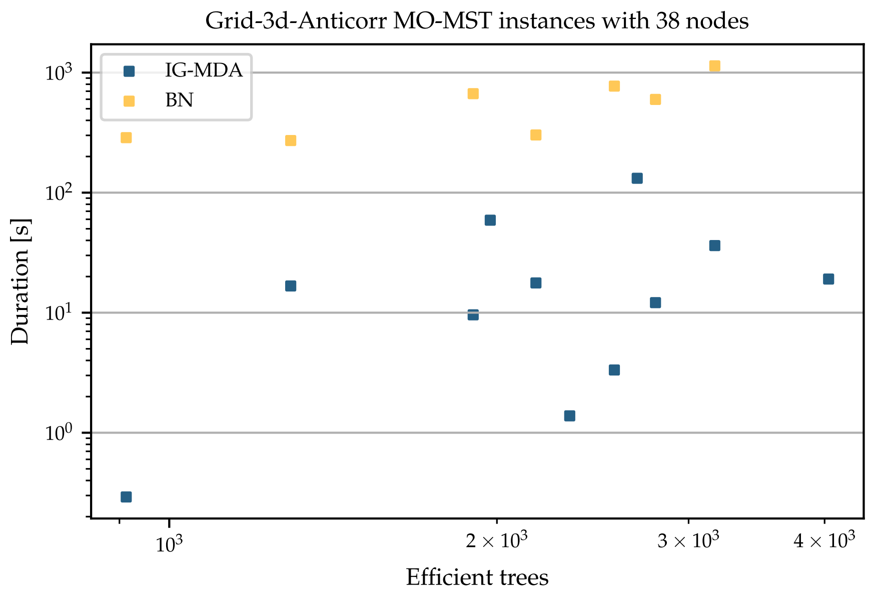


Figure 35: Running times on Grid-3d-Anticorr instances with 38 nodes.

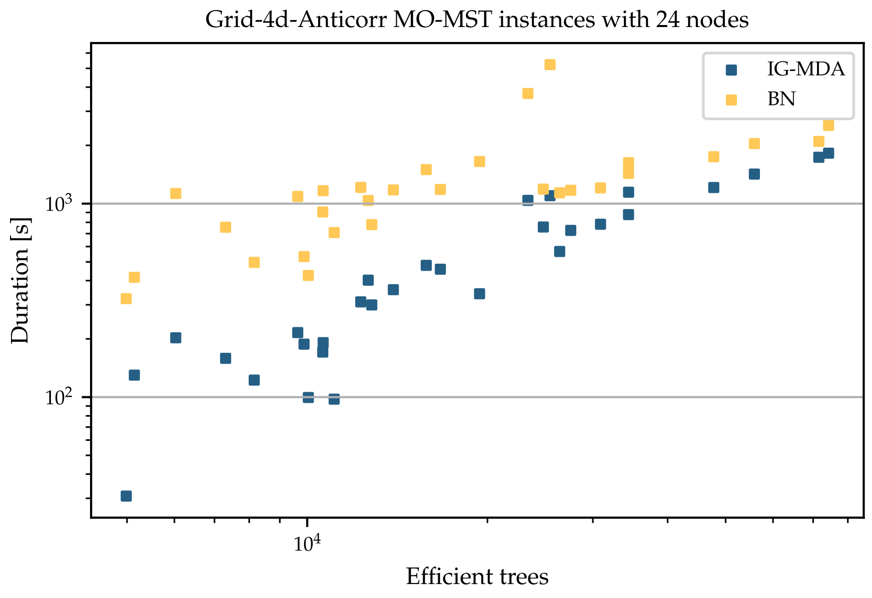


Figure 36: Running times on Grid-4d-Anticorr with 24 nodes.

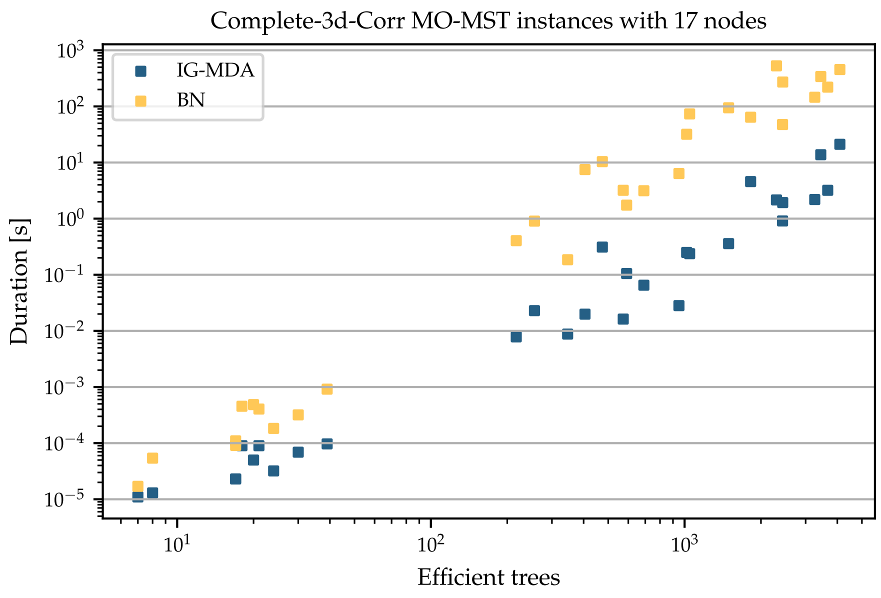


Figure 37: Running times on Complete-3d-Corr instances with 17 nodes.

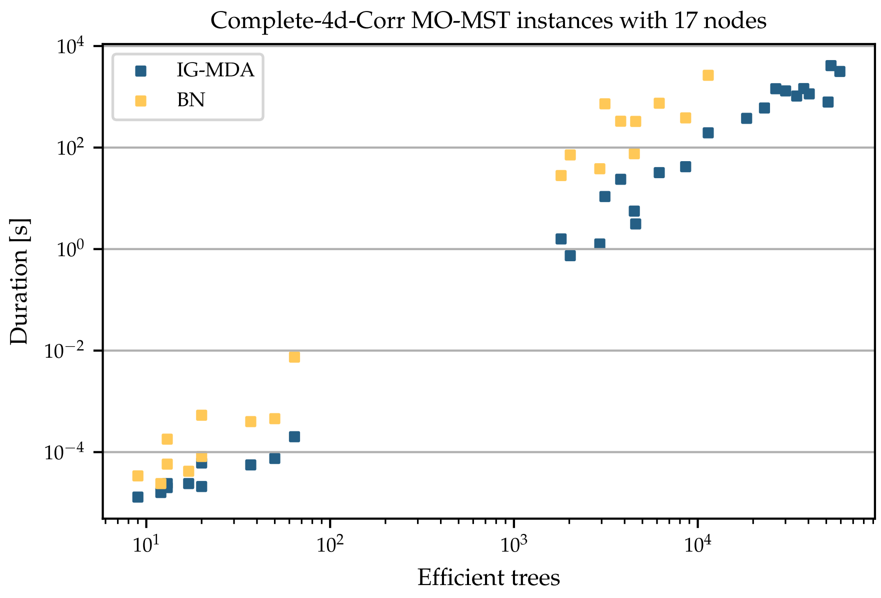


Figure 38: Running times on Complete-4d-Corr instances with 17 nodes.

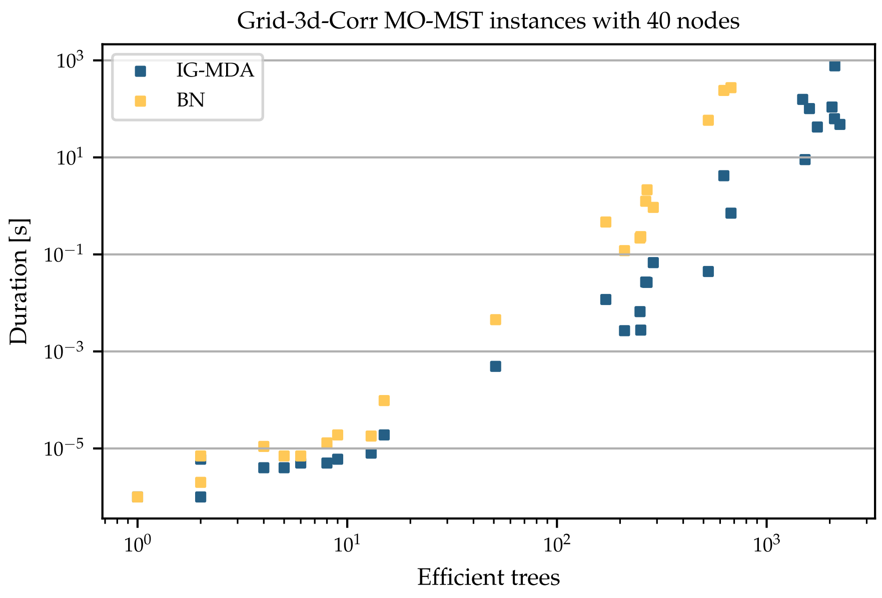


Figure 39: Running times on Grid-3d-Corr instances with 40 nodes.

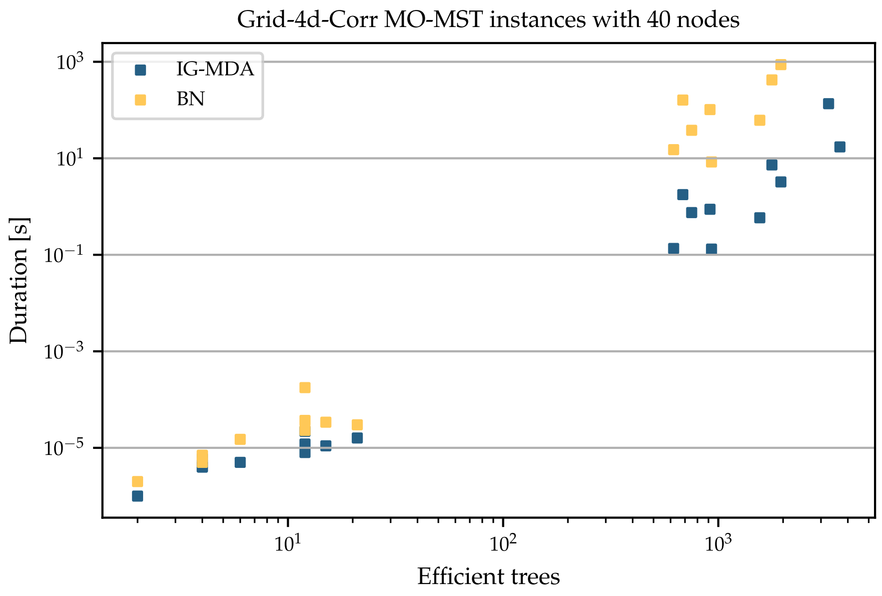


Figure 40: Running times on Grid-4d-Corr instances with 40 nodes.

$\|V\|$	Insts	BN					IG MDA					Speedup
		Sol	$\|T^*_G\|$	$\|\mathcal{V}\|$	Iterations	Time	Sol.	$\|T_G\|$	$\|\mathcal{V}\|$	Iterations	Time	
						COMPLETE 3d anticorr. edge costs						
10	30	30	1934.17	509.62	79059.14	0.4	30	1934.17	477.88	58042.67	0.08	4.82
12	30	30	4042.89	2039.89	668348.06	7.68	30	4042.89	1868.86	434829.76	1.27	6.03
15	30	30	10507.52	16344.5	13781596.11	530.79	30	10507.6	15166.2	8203636.87	73.52	7.22
17	30	10	8995.82	65151.83	39926366.12	4545.6	29	16773.63	60128.77	50138867.03	775.91	10.22
						GRID 3d anticorr. edge costs						
18	30	30	327.46	1970.48	27559.31	0.0444	30	327.46	1059.14	15957.42	0.0105	4.24
20	30	30	507.80	4321.96	80646.78	0.1788	30	507.80	2081.46	41301.82	0.0354	5.05
24	30	30	1905.73	289945.96	12561838.13	119.8732	30	1905.73	108148.81	4556890.58	19.8963	6.02
27	30	25	2055.18	658592.00	32710242.90	331.8889	30	2432.92	243115.52	13189507.15	77.9690	7.27
30	30	8	1400.97	798812.92	32607409.69	336.6812	20	2119.93	238092.28	13277318.30	75.8248	31.61
33	30	2	2091.47	1353679.98	68100519.74	728.4824	16	3285.21	637610.27	44246349.41	341.4173	26.16
36	30	0	–	–	–	–	10	2945.52	497406.55	34694216.43	263.5790	–
38	30	7	1959.45	938641.43	51235269.92	504.1519	11	2189.33	35140.31	2977740.76	10.8611	68.16
40	30	0	–	–	–	–	2	3440.77	1310244.04	98125888.60	874.0739	–
						COMPLETE 4d anticorr. edge costs						
10	30	30	14261.89	511.50	275211.78	3.61	30	14261.89	499.37	242326.48	0.85	4.27
12	30	30	49937.31	2046.63	3635468.74	266.18	30	49937.32	2006.20	2951674.07	31.32	8.50
15	30	0	–	–	–	–	22	200339.04	16224.95	77909983.48	4468.7791	–
						GRID 4d anticorr. edge costs						
18	30	30	1605.54	3338.71	83063.06	0.1761	30	1605.54	2548.04	65344.60	0.0742	2.37
20	30	30	2022.36	6941.00	211043.31	0.5899	30	2022.36	4681.20	144836.59	0.2144	2.75
24	30	30	17036.10	487336.31	56591949.94	1145.0088	30	17036.10	308131.18	35528444.71	382.9531	2.99
27	30	12	10091.80	830056.49	70224110.93	1539.2769	30	21918.58	606263.90	83839892.33	1092.2398	5.93
30	30	0	–	–	–	–	8	21205.22	955448.68	133902354.68	2850.7280	–
33	30	0	–	–	–	–	2	13126.77	1776005.39	207538764.46	3671.7716	–

Table 18: Instances from (Fernandes et al., 2020) with anticorrelated edge costs. Cardinality $|T_G|$ of solution sets, $|\mathcal{V}|$ of transition nodes, number of iterations, and time report geometric means built among solved instances only. The speedups consider only instances solved by both algorithms.

The results from the instances with correlated edge cost functions are reported in Table 19. As expected, the size of the solved instances is bigger than in the anticorrelated counterparts. However, this effect can be misleading. As we can observe in the scatter plots in Figure 37 to Figure 40 the running times of both algorithms are separated into two clusters. The reason is that with correlated costs, the preprocessing phase sometimes recognizes many blue and red arcs (cf. Section 13.2.3) s.t. the input graph for the MO-MST algorithms is actually very small. In fact, there are even 3d grid instances with 36 nodes that contain 35 blue arcs s.t. the remaining graph consists only of one node. The 35 blue arcs constitute the only efficient spanning tree for this instance. On grid graphs with a fixed number of nodes, the running times of both algorithms can differ by more than eight orders of magnitude (cf. Figure 40). Therefore, the geometric means in Table 19 are almost always clearly below one second but they need to be put into the context described in this paragraph. Only then, given the low average running times, we can understand that not all instances are solved.

COMPLETE GRAPHS WITH CORRELATED COSTS On 3d instances on complete graphs (Table 19), both algorithms solve all instances with up to 17 nodes. On these instances the IG-MDA is $\times 29.09$ faster (Figure 37). The IG-MDA also solves all instances with 22 nodes in 1.06s on average. In this group of instances, the BN algorithm solves 19/30 instances. Even though the edge costs are correlated, the instances on complete graphs with 4d edge costs are difficult and the BN algorithm cannot solve all instances with 15 nodes. On these graphs, the speedup in favor of the IG-MDA is $\times 15$; higher than on the same graphs with 3d edge costs.

GRID GRAPHS WITH CORRELATED COSTS The lower left clusters in Figure 39 and Figure 40 show that indeed a considerable amount of instances in this group are almost solved during preprocessing. After deleting red edges and contracting blue ones, the remaining instances are so small that both MO-MST algorithms solve them in less than a millisecond. Overall, the IG-MDA remains consistently more than an order of magnitude faster than the BN algorithm on this type of instances (cf. Table 19). After deleting red edges and contracting blue edges, we are left with 12 instances with 4d edge costs and 30-31 nodes. To the best of our knowledge these instances are the biggest considered in the literature so far and the IG-MDA solves them in 329.9s on average (geo. mean). This data is extracted from the file ../results/multiPrim_grid_corr_4d.csv in (Maristany de las Casas, 2023c).

$\lvert V \rvert$	Insts	BN					IG MDA					Speedup
		Sol	$\lvert T^*_G \rvert$	$\lvert \mathcal{V} \rvert$	Iterations	Time	Sol.	$\lvert T_G \rvert$	$\lvert \mathcal{V} \rvert$	Iterations	Time	
COMPLETE 3d corr. edge costs												
15	30	30	195.18	786.77	16508.49	0.0438	30	195.18	324.67	4988.89	0.0038	11.50
17	30	30	276.97	3240.53	101568.59	0.4329	30	276.97	769.99	14026.10	0.0149	29.09
20	30	22	289.40	6793.78	186354.27	0.7750	30	626.75	4364.52	109898.99	0.2118	35.26
22	30	19	228.20	6070.89	154896.66	0.5645	30	847.30	10209.11	390923.30	1.0635	24.93
GRID 3d corr. edge costs												
33	30	30	77.65	915.59	7991.91	0.0185	30	77.65	176.45	2042.85	0.0013	14.42
36	30	28	76.38	1521.15	15417.74	0.0808	30	93.14	342.61	4519.75	0.0037	18.52
38	30	30	56.21	396.92	3136.16	0.0060	30	56.21	89.00	966.31	0.0005	12.56
40	30	21	39.58	349.33	2406.14	0.3498	29	114.07	848.70	10929.89	0.0180	12.46
COMPLETE 4d corr. edge costs												
12	30	30	326.82	283.94	7713.31	0.0319	30	326.82	176.10	4076.06	0.0057	5.64
15	30	26	713.27	1646.49	81809.87	0.7962	30	1177.38	994.76	48608.53	0.1733	14.99
17	30	20	294.13	1191.90	31441.59	0.1965	30	1448.65	2110.19	124335.61	0.7096	11.50
20	30	13	305.19	2523.53	56573.88	0.1986	21	1444.82	3938.87	155447.82	0.6001	25.25
GRID 4d corr. edge costs												
24	30	30	89.11	690.09	5327.06	0.0135	30	89.11	221.65	2098.39	0.0020	6.79
27	30	30	167.53	1882.84	20551.53	0.0823	30	167.53	446.02	5610.28	0.0072	11.50
30	30	28	163.55	2795.94	35350.59	0.1349	30	216.40	840.08	13791.31	0.0216	13.05
33	30	23	82.51	669.94	5933.57	0.0141	29	206.47	842.73	14760.35	0.0222	9.27
36	30	20	49.51	311.78	2518.64	0.0057	27	193.85	812.96	13934.90	0.0225	11.20
38	30	26	144.41	806.94	11284.06	0.0347	30	235.57	352.92	8144.57	0.0112	11.46
40	30	18	66.30	598.21	4758.39	0.0153	20	98.46	234.39	3118.01	0.0036	12.27

Table 19: Instances from (Fernandes et al., 2020) with correlated edge costs. Cardinality $\lvert T_G \rvert$ of solution sets, $\lvert \mathcal{V} \rvert$ of transition nodes, number of iterations, and time report geometric means built among solved instances only. The speedups consider only instances solved by both algorithms.

13.6 CONCLUSION

The Implicit Graph Multiobjective Dijkstra Algorithm (IG-MDA) is a new Dynamic Programming algorithm for the Multiobjective Minimum Spanning Tree (MO-MST) problem. For its design we manipulated the *transition graph* defined in Santos et al. (2018) using cost-dependent criteria to reduce its size and thus enhance the performance of the used algorithms. Dynamic Programming for MO-MST problems entails solving an instance of the One-to-One Multiobjective Shortest Path problem defined on the arising transition graph. In this chapter, we analyzed the size of the transition graph to motivate its implicit handling in the IG-MDA. Storing the graph explicitly would otherwise lead to unreasonable memory consumption on many instances. To benchmark the performance of the IG-MDA, we compared it to a new version of the BN algorithm from (Santos et al., 2018) on a big set of instances from the literature. The results show that the IG-MDA is more efficient. To the best of our knowledge, it also solves bigger MO-MST instances than the biggest ones solved so far in the literature.

14 K-SHORTEST PATH PROBLEM

14.1 INTRODUCTION

The k-*Shortest Simple Path* problem is a well known optimization problem. With the background from other chapters in this thesis, we can define it right away formally.

Definition 14.1 (k-Shortest Simple Path Problem)**.** Given a directed graph $D = (V, A)$, two nodes s, $t \in V$, an arc cost function $c : A \to \mathbb{R}_{\geqslant 0}$, and an integer $k \geqslant 2$, let P_{st} be the set of simple s-t-paths in D. Assume P_{st} contains at least k paths. The k-*Shortest Simple Path* (k-SSP) problem is to find a sequence $P = (p_1, p_2, \ldots, p_k)$ of pairwise distinct s-t-paths with $c(p_i) \leqslant c(p_{i+1})$ for any $i \in \{1, \ldots, k-1\}$, s.t. there is no path $p \in P_{st} \setminus P$ with $c(p) < c(p_k)$. We refer to the tuple $\mathcal{I} := (D, s, t, c, k)$ as a k-SSP instance and call P a solution sequence.

It is worth noting that the problem is defined on weighted directed graphs, i.e., an arc's cost is a scalar in contrast to the vectors of costs considered so far in this thesis. At first, skeptical readers might wonder what this scalar optimization problem has in common with the multiobjective algorithms described used in previous chapters. The answer is that in our new k-SSP algorithm we can use a BOPS algorithm as a subroutine that is called $\mathcal{O}(k)$. Given the scalar arc costs, we define a second arc cost component on every arc. It is used to discard shortest subpaths (w.r.t. the original costs) found during the search for a j^{th} shortest path, $j \in \{1, \ldots, k\}$, that are not simple. With this additional cost component, we can identify subpaths with cycles with only one $\preceq_D$ -check. Using the dimensionality reduction technique (cf. Section 3.5) and given that the new arc costs are bidimensional, the $\preceq_D$-check consists of just one comparison. Hence, in our setting we are able to discard non-simple subpaths without the need to store any information about their nodes.

The skeptical readers, even if convinced by the appealing possibility of checking in $\mathcal{O}(1)$ whether a path is simple, might argue that in every of the $\mathcal{O}(k)$ calls to our BOSP algorithm as a subroutine, a minimal complete set of efficient paths needs to be computed and the output can have exponential size. This concerns are justified but our new second arc cost component guarantees that at most $\mathcal{O}(n)$ paths are contained in a minimal complete set of efficient paths. This polynomial bound on the output size of our BOSP subroutine allows us to derive an asymptotic running time bound for our new k-SSP algorithm that matches the state of the art.

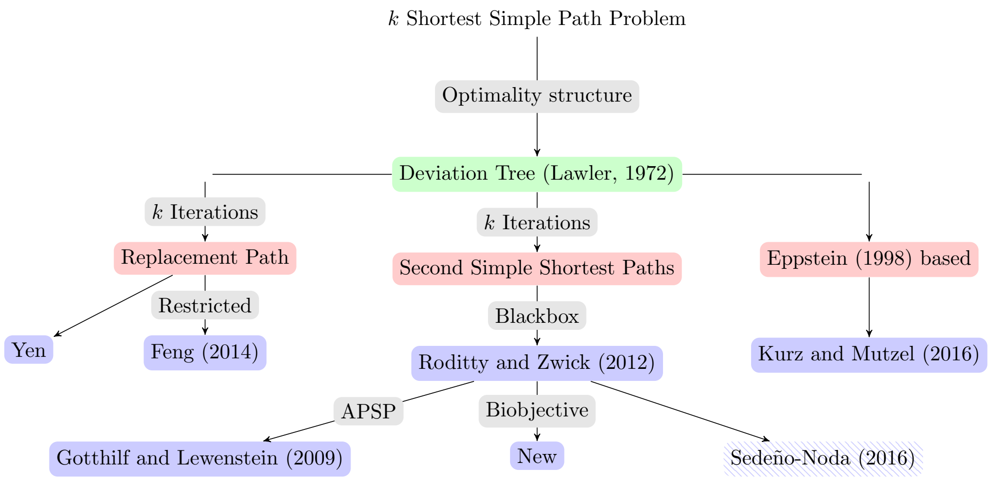


Figure 41: Relevant literature for this chapter. The green node represents the optimality structure used by all algorithms to keep track of the solution paths and to avoid duplicates (see Section 14.2). Red nodes symbolize solution approaches. Blue nodes refer to k-SSP algorithms. If a blue node has solid background, the corresponding algorithm has a state of the art asymptotic running time.

14.1.1 Literature Overview

The oldest reference on the k-SSP we could find in the literature is the work by Clarke et al. (1963). A detailed literature survey on this topic is given by Eppstein (2016). Figure 41 gives a visual overview of publications that are relevant in this chapter. The figure serves also as an outline for this section.

To solve the k-SSP problem efficiently, algorithms need to keep track of the s-t-paths found so far and be able to avoid the generation of duplicates without the need to pairwise compare a new path with every existing path. All relevant k-SSP algorithms do so using an *optimality structure* called *deviation tree* first devised by Lawler (1972), to be discussed in Section 14.2. It is based on the consideration of subpaths.

Definition 14.2. Given a digraph $D = (V, A)$ and a simple u-w-path p in D between distinct nodes $u, w \in V$, we denote a subpath of p between nodes $v, v' \in p$ by $p^{v \to v'}$. Thereby if $v = s$ we call $p^{v \to v'} = p^{s \to v'}$ a *prefix* of p and if $v' = t$ we call $p^{v \to v'} = p^{v \to t}$ a *suffix* of p.

The classical k-SSP algorithm is due to Yen (1972). It performs k iterations and starts with a solution sequence $P = (p_1)$ containing only a shortest s-t-path p_1. In the ith iteration, $i \in \{1, \dots, k\}$, an ith shortest s-t-path p_i is considered and the following set of s-t-paths is computed.

$$\left\{q \;\middle|\; q = p_i^{s \to v} \circ q^{v \to t},\ q^{v \to t} \text{ shortest } v\text{-}t\text{-path in } D \setminus \{\delta^+(v) \cap p_i\},\ v \in p_i\right\}. \tag{43}$$

The set is a solution to the so called *Replacement Path* (RP) problem. The paths from this set and from all such sets computed in previous iterations are stored in a priority queue of s-t-paths from which, at the beginning of the $(i+1)$th iteration, a $(i+1)$th shortest path is extracted and stored in the solution sequence P. The simple s-t-path p_i has at most n nodes and thus,

(43) contains at most $n-1$ elements, each of them requiring a shortest path computation to obtain the suffix $q^{\nu \to t}$. Yen's algorithm solves the RP instances in a straightforward way iterating over the nodes in p_i and solving the corresponding Shortest Path instances. Using Dijkstra's algorithm (Dijkstra, 1959) with a Fibonacci Heap (Fredman & Tarjan, 1987) for these queries, we obtain a running time for the solution of the RP problem of

$$T_{RP} := \mathcal{O}\left(n(n \log n + m)\right). \tag{44}$$

The deviation tree by Lawler (1972) (also called *pseudo-tree* in the literature (cf. E. Q. V. Martins & Pascoal, 2003)) is used to ensure that the solution paths for (43) computed in every iteration of Yen's algorithm differ from each other without the need to pairwise compare them. Then, Yen's algorithm has an asymptotic running time of

$$\mathcal{O}\left(kT_{RP}\right). \tag{45}$$

There is a recent alternative k-SSP algorithm running in $\mathcal{O}\left(kT_{RP}\right)$ that can also be considered the current state of the art. It is due to Kurz and Mutzel (2016). We refer to this algorithm as the *KM algorithm*. Interestingly, the authors derive this running time bound even if the KM algorithm does not solve the RP problem as a subroutine. Instead, their algorithm can be seen as a generalization of Eppstein's algorithm (Eppstein, 1998) for the k-Shortest Path problem in which the output paths are allowed to contain cycles. Instead of solving One-to-One Shortest Path instances as required in (43), the KM algorithm solves One-to-All Shortest Path instances, hence obtaining a shortest path tree from every search. These instances are defined on the reversed input digraph and are rooted at the target node. The main idea of the KM algorithm, similar to the idea in (Eppstein, 1998), is that $\mathcal{O}(m)$ simple s-t-paths can be obtained from such a tree using non-tree arcs to create alternative s-t-paths. By doing so, a cycle may be constructed in which case the KM algorithm needs to compute a new shortest path tree. In addition to its state of the art running time bound, the efficiency of the KM algorithm in practice is immediately apparent: in *well behaved* networks, only few shortest path trees are needed since the swapping of tree arcs and non-tree arcs yields enough simple s-t-paths. Indeed, in the computational experiments conducted in (Kurz & Mutzel, 2016), the KM algorithm clearly outperforms the previous state of the art k-SSP algorithm by Feng (2014a). This algorithm resembles Yen's algorithm but partitions nodes into three classes, being able to ignore nodes from one of the classes while solving (43). Due to the reduced search space/graph, the One-to-One Shortest Path computations finish faster than in Yen's algorithm.

Better Asymptotics and Better Computational Performance

The algorithm by Gotthilf and Lewenstein (2009) (*GL algorithm*) improves the best known asymptotic running time for the k-SSP problem. It uses the All Pairs Shortest Path (APSP) algorithm from Pettie (2004) to achieve an asymptotic running time bound of $\mathcal{O}\left(k(n^2 \log\log n + nm)\right)$. Here, the term $(n^2 \log\log n + nm)$ corresponds to the APSP running time bound derived by Pettie and k APSP instances need to be solved. As a brief digression from the main focus of the chapter, we remark that a new APSP algorithm published in Orlin and Végh (2022) achieves an asymptotic running time bound of $\mathcal{O}(mn)$ for instances with nonnegative integer arc costs. Using this new algorithm as a subroutine in the GL algorithm, the following result is immediate.

Theorem 14.1 (k-SSP Running Time for Integer Arc Costs)**.** *The* k*-SSP problem from Definition 14.1 with integer arc costs can be solved in* $\mathcal{O}(kmn)$ *time.*

Despite the unbeaten asymptotic running time bound, the GL algorithm does not perform well in practice. Solving k APSP instances requires too much computational effort.

There are k-SSP algorithms whose asymptotic running time bound is worse than (45) and possibly not even pseudo-polynomial but they perform extremely well in practice. The current state of the art among these algorithms is published in Sedeño-Noda (2016) and in Feng (2014b), the latter publication being based on the MPS algorithm (E. d. Q. V. Martins et al., 1999). Both algorithms are very different from the ones we study here and we choose to analyze k-SSP algorithms that run in (45).

14.1.2 Contribution and Outline

Figure 41 shows that there is a third approach to the k-SSP problem. Namely, Roditty and Zwick (2012, 2005) show that the k-SSP problem can be tackled by solving at most $2k$ instances of the *Second Simple Shortest Path* (2-SSP) problem. In their publications, the authors do not specify how the 2-SSP instances arising as subproblems in their algorithm can be solved efficiently.

We design, for the first time, a computationally competitive version of the black box algorithm by Roditty and Zwick. To do so we use a novel algorithm for the 2-SSP problem. This algorithm is based on a One-to-One version of the recently published Biobjective Dijkstra Algorithm (BDA) (Sedeño-Noda & Colebrook, 2019) and incorporates the One-to-One MOSP techniques discussed in Chapter 9.

In Section 14.2 we describe the *deviation tree*, the optimality structure used throughout the chapter. In Section 14.3 we discuss our main contribution: a new 2-SSP algorithm using a biobjective approach. In Section 14.4 we describe the k-SSP algorithm by Roditty and Zwick (2012) that solves $\mathcal{O}(k)$ 2-SSP instances. In Section 14.5 we demonstrate the efficiency of our algorithm in practice, comparing it with the KM algorithm (Kurz & Mutzel, 2016).

14.2 OPTIMALITY STRUCTURE — DEVIATION TREE

Consider a k-SSP instance $\mathcal{I} := (D, s, t, c, k)$. A (partial) solution sequence $P_\ell = (p_1, \dots, p_\ell)$ for $\ell \in \{1, \dots, k\}$ is represented as a *deviation tree* T_{P_ℓ} (e.g., Roditty & Zwick, 2012; Lawler, 1972; E. Q. V. Martins & Pascoal, 2003). T_{P_ℓ} is a directed graph, represented as a tree in which a node from the original graph D may appear multiple times. The root node of T_{P_ℓ} is a copy of the node s in D and every leaf corresponds to a copy of the node t. There are

ℓ leafs and any path from the root to a leaf is in one-to-one correspondence with an s-t-path in D.

Definition 14.3 (Deviation Tree)**.** The deviation tree T_{P_ℓ} is built iteratively. Initially, T_{P_ℓ} is empty. p_1 is added to T_{P_ℓ} by adding all nodes and arcs of p_1 to the tree. For any $j \in \{2, \dots, \ell\}$, assume that the previous paths p_i, $i < j$ have been added to T_{P_ℓ} already. Assume the longest common prefix of p_j with a path $p \in P_{j-1}$ is the s-v-subpath $p_j^{s \to v}$ for a node $v \in p_j$. Then, p_j is added to T_{P_ℓ} by appending its suffix $p_j^{v \to t}$ to the copy of v along p in T_{P_ℓ}.

PARENT PATH The path p_1 has no parent path. For any path p_i, $i \in \{2, \dots, \ell\}$, its *parent path* p is the path in P_ℓ with which p_i shares the longest (w.r.t. the number of arcs) common prefix $p_i^{s \to v}$. In case p is not uniquely defined, p is set to be the first path in P_ℓ with $p^{s \to v} = p_i^{s \to v}$. If p is the parent path of p_i, p_i is a *child path* of p.

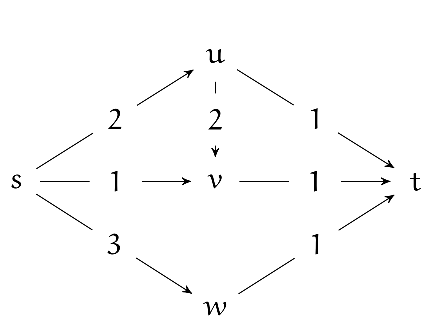
DEVIATION ARC, DEVIATION NODE, SOURCE NODE The path p_1 has no *deviation arc*, its *deviation node* is s and it its *source node* is also s. For any path p_i, $i \in \{2, \dots, \ell\}$ its *deviation arc* is the first arc (v, w) along p_i after the common prefix of p_i with its parent path. The node v is called the *deviation node* of p_i and the node w is called the *source node* of p_i. For any path $p \in P_\ell$ we write $\mathrm{dev}(p)$ and $\mathrm{source}(p)$ to refer to these nodes.

Recall that we assume the paths in P_ℓ to be sorted non-decreasingly according to their costs. Then, the parent path p of any path $q \in P_\ell$ is stored before q in P_ℓ and we have $c(p) \leqslant c(q)$. Moreover, the inductive nature of T_{P_ℓ} guarantees that, along p, the deviation node of p does not come after the deviation node of q.

Example 14.1. The left hand side of Figure 42 shows a 4-SSP instance. We set $P = (p_1, \dots, p_4)$ with $p_1 = ((s, v), (v, t))$, $p_2 = ((s, u), (u, t))$, $p_3 = ((s, w), (w, t))$, and $p_4 = ((s, u), (u, v), (v, t))$. The right hand side depicts the deviation tree T_P as defined in Definition 14.3. When building T_P iteratively, p_1 is added first. Then, the longest common prefix of p_1 and p_2 is identified to be just the node s. Thus, p_2 is appended to s in T_P. The parent path of p_2 is p_1, the deviation node is s, and the source node is u. Adding p_3 to T_P leads to the situation in which two paths, p_1 and p_2, share the longest common prefix with p_3: the node s only. Hence, the parent path of p_3 is set by definition to be p_1: the first path in P sharing the longest common prefix with p_3. p_3's deviation node is s and its source node is w. Finally, the path p_4 is added to T_P. It shares its prefix $p_4^{s \to u}$ of maximum length with p_2 and thus, its suffix $p_4^{u \to t} = ((u, v), (v, t))$ is appended in T_P to the copy of the node u along p_2 in T_P. The deviation node of p_4 is u; its source node is v.

14.3 SECOND SHORTEST SIMPLE PATH PROBLEM

We introduce a new 2-SSP algorithm. Assuming that a shortest path p is known, we define a biobjective arc cost function $\gamma_p : A \to \mathbb{R}^2$ depending on p. Using γ we can find a second shortest simple path as the first or the second (in lexicographic order w.r.t. γ) efficient solution of a One-to-One BOSP instance associated with p.



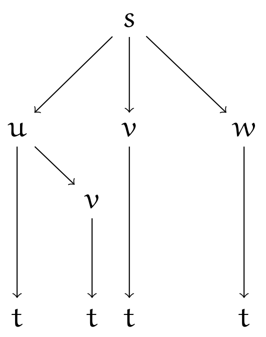


Figure 42: Left: Input graph D and arc costs for a k-SSP instance. Right: Pseudo-Tree T_P corresponding to the k-SSP instance defined on the left with $k = 4$.

Definition 14.4. Consider a digraph $D = (V, A)$, nodes $s, t \in V$, and an arc cost function $c : A \to \mathbb{R}_{\geqslant 0}$. $\mathcal{I} = (D, s, t, c)$ is an instance of the classical One-to-One Shortest Path problem. Let p be a shortest s-t-path in D. For every arc $a \in A$ we define bidimensional costs $\gamma_p(a) \in \mathbb{R}^2_{\geqslant 0}$ setting $\gamma_{p,1}(a) = c(a)$ and $\gamma_{p,2}(a) = 1$ iff $a \in p$. Otherwise, we set $\gamma_{p,2}(a) = 0$. The One-to-One BOSP instance $\mathcal{I}^p_{\text{BOSP}} := (D, s, t, \gamma_p)$ is the *BOSP instance associated with $\mathcal{I}$ and* p.

The γ_p function defined in Definition 14.4 is such that the biggest minimal complete set of efficient paths in $\mathcal{I}^p_{\text{BOSP}}$ contains at most $(n-1)$ paths (cf. Lemma 14.1). This makes the $\mathcal{I}^p_{\text{BOSP}}$ instances tractable. Moreover, the $\{0,1\}$ second component of γ_p plays an essential role to circumvent the issue of second shortest paths not adhering to the subpath-optimality principle as explained in the following example.

Example 14.2. Consider the $\mathcal{I}^p_{\text{BOSP}}$ instance defined in Figure 43 w.r.t. the shortest s-t-path p in that instance. The shown graph contains two s-v_4-paths:

$$q = ((s, v_1), (v_1, v_4)) \text{ with } c(q) = \gamma_{p,1}(q) = 2$$
$$r = ((s, v_1), (v_1, v_2), (v_2, v_3), (v_3, v_4)) \text{ with } c(r) = \gamma_{p,1}(r) = 1$$

A label-setting Shortest Path algorithm would discard q since $c(r) < c(q)$. However, the extension of r towards v_2 produces a cycle, causing any expansion of r to be an invalid candidate for a second shortest simple s-t-path. Thus, when comparing q and r both paths need to be recognized as *promising* candidates. Since q shares only one arc with p before it deviates, we have $\gamma_{p,2}(q) = 1$. Additionally, we have $\gamma_{p,2}(r) = 3$. Thus, $\gamma_p(q) = (2, 1)$ and $\gamma_p(r) = (1, 3)$ and both paths are efficient/optimal s-v_4-paths in our biobjective setting.

Note that v_2 is already visited by r's subpath $r^{s \to v_2}$ with costs $\gamma_p(r^{s \to v_2}) = (0, 2)$. After expanding q and r along the arc (v_4, v_2), we have $\gamma_p(q \circ (v_4, v_2)) = (4, 1)$ and $\gamma_p(r \circ (v_4, v_2)) = (3, 3)$ and we see that r's expansion is dominated by $r^{s \to v_2}$ and thus can be discarded. q's expansion on the other side is not dominated and thus, the *bad* s-v_4-path w.r.t. the original cost function c is kept to build a simple second shortest s-t-path.

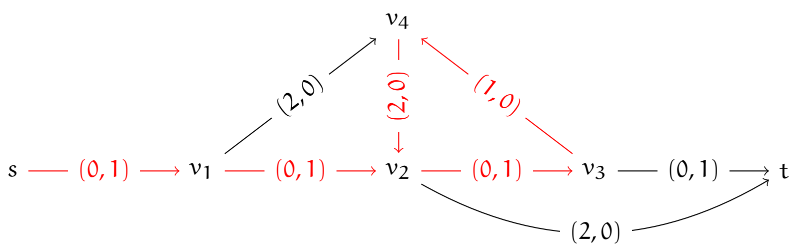


Figure 43: $\mathcal{I}_{BOSP}^{p}$ instance for the shortest path $p = ((s,v_1),(v_1,v_2),(v_2,v_3),(v_3,t))$. The red arcs show a s-v_2-path that is not simple. Its s-v_4-subpath is the shortest s-v_4-path w.r.t. the original arc costs $c = \gamma_1$.

The γ arc cost function not only elevates the status of paths with suboptimal subpaths w.r.t. c to become efficient paths in the $\mathcal{I}_{BOSP}$ instances. Using the *dimensionality reduction* technique in these biobjective scenarios, we can perform $\preceq_D$ -checks in constant time (cf. Corollary 3.2). Since, as in the last example, paths that are not simple turn out to be dominated, we manage to detect cycles in constant time and without the need to hash a path's nodes.

To solve the $\mathcal{I}_{BOSP}$ instances and obtain a running time bound for our k-SSP algorithm in Section 14.4 that is on par the running time bound from Yen's algorithm, we use the biobjective version of the MDA that was originally called the *Biobjective Dijkstra Algorithm* and published in (Sedeño-Noda & Colebrook, 2019).

14.3.1 Second Simple Shortest Paths Using the BDA

We formulate the following main result in this section.

Theorem 14.2. *Consider a shortest path problem $\mathcal{I} = (D,s,t,c)$, let p be a shortest s-t-path w.r.t. c and assume it has ℓ arcs. A lexicographically smallest (w.r.t. γ_p) efficient s-t-path q with $\gamma_{p,2}(q) < \ell$ in the BOSP instance $\mathcal{I}_{BOSP}^{p}$ is a second shortest simple s-t-path in D w.r.t. the original costs c.*

Proof. We assume that $\mathcal{I}_{BOSP}^{p}$ is solved using the BDA. Every efficient path in this instance is a simple path or cost-equivalent to a simple path since γ_p is a non-negative function. Additionally, efficient paths containing a loop are neither made permanent nor further expanded by the BDA. As a consequence, every path q that is made permanent fulfills $\gamma_{p,2}(q) \leqslant \ell$ and the only possibly extracted path with $\gamma_{p,2}(q) = \ell$ is p with costs $\gamma_p(p) = (c(p), \ell)$.

Since p is a shortest s-t-path, every extracted s-t-path $q \neq p$ fulfills $\gamma_{p,1}(q) \geqslant \gamma_{p,1}(p)$. Thus if q is made permanent before p, it is lex-smaller than p and we must have $\gamma_{p,1}(q) = \gamma_{p,1}(p)$ and $\gamma_{p,2}(q) < \gamma_{p,2}(p)$. The second inequality implies that q and p are distinct paths and we thus can stop the execution of the BDA and return q as a second shortest simple s-t-path. In this case, p and q are cost-equivalent w.r.t. c.

Assume p is permanent already and q is the next s-t-path extracted from the BDA's priority queue. We must have $\gamma_{p,1}(p) < \gamma_{p,1}(q)$ (see last paragraph). Recall that the BDA finds a minimal complete set of efficient paths for $\mathcal{I}_{BOSP}^{p}$. Moreover, as already noted, paths are extracted from the algorithm's priority queue in lex. nondecreasing order. Thus, since efficient paths are simple, we conclude that there cannot exist a simple s-t-path q' with $\gamma_{p,1}(p) \leqslant \gamma_{p,1}(q') < \gamma_{p,1}(q)$ that is not found by the BDA. Since the $\gamma_{p,1}$ costs are equivalent to the original c costs of the paths in D, we obtain that q is a second shortest simple s-t-path. □

The shortest path p is the only simple path in $\mathcal{I}^p_{BOSP}$ with $\gamma_2(p) = \ell$. Since we want to find an efficient s-t-path q with $\gamma_{p,2}(q) < \ell$, the BDA can stop after at most two s-t-paths are extracted from the priority queue: the path p that is efficient iff $\gamma_{p,1}(p) < \gamma_{p,1}(q)$ and q itself. Using this stopping criterion, we define the following modified version of the BDA as our new 2-SSP algorithm.

Definition 14.5 (BDA$_{2SSP}$)**.** As so far in this section, consider the 2—SSP instance $\mathcal{I}^p_{BOSP}$ associated with k—SSP instance $\mathcal{I}$ and a path p. The BDA$_{2SSP}$ solves $\mathcal{I}^p_{BOSP}$. It modifies the BDA as follows.

INPUT In addition to a BOSP instance, the input of the BDA$_{2SSP}$ contains a non-negative integer ℓ.

STOPPING CONDITION The BDA$_{2SSP}$ stops whenever an efficient s-t-path q with $\gamma_{p,2}(q) < \ell$ is extracted from the priority queue or when the priority queue is empty at the beginning of an iteration.

OUTPUT Instead of a minimal complete set of efficient s-t-paths, the new BDA$_{2SSP}$ returns the suffix $q^{v \to t}$ of the first efficient s-t-path q with $\gamma_{p,2}(q) < \ell$ that it finds. Here, v is the node after which p and q deviate for the first time. If such a path q does not exist, the BDA$_{2SSP}$ returns a dummy path if such a path does not exist.

It is easy to see that Theorem 14.2 proves the correctness of the BDA$_{2SSP}$. We can thus move on to the analysis of its asymptotic behavior.

14.3.2 Asymptotic Running Time and Memory Consumption

The following is a general statement that holds for any biobjective optimization problem. We cannot find it explicitly stated in the literature but it is certainly not our contribution. Recently it has been used implicitly in e.g., (Gorski et al., 2022).

Lemma 14.1. *Let $\mathcal{X}$ be the set of feasible solutions of a biobjective optimization problem and $f : \mathcal{X} \to \mathbb{R}_{\geqslant 0} \times \mathbb{N}$ the associated cost function. The cardinality of a minimal complete set of efficient solutions is bounded by the size of the set $\{f_2(x) \mid x \in \mathcal{X}\}$.*

Proof. Assume for a value $y_2 \in \{f_2(x) \mid x \in \mathcal{X}\}$ there are two efficient solutions x, x' in a minimal complete set. If $f_1(x) \neq f_1(x')$, then the solution with the smaller f_1 value (weakly) dominates the other. If $f_1(x) = f_1(x')$, both solutions are cost-equivalent and, by definition, no minimal complete set contains both. □

Our setting in this section assumes a shortest s-t-path p with ℓ arcs to be given and we look for a second shortest s-t-path in the same graph. In the first paragraph of the proof of Theorem 14.2 we derived that any efficient path q for the instance $\mathcal{I}^p_{\text{BOSP}}$ fulfills $\gamma_{p,2}(q) \leqslant \ell$. Lemma 14.1 applied to $\mathcal{I}^p_{\text{BOSP}}$ implies that every minimal complete set of efficient s-v-paths, $v \in V$, has cardinality at most ℓ.

For the set of efficient s-t-paths computed by the BDA$_{2\text{SSP}}$ we even now that it contains at most two paths at the end of the algorithm. Sadly, we cannot mirror this fact in the running time bound of the BDA$_{2\text{SSP}}$. As explained in Chapter 9, in One-to-One BOSP instance, a minimal complete set of efficient s-t-paths can contain less paths than the number of efficient s-v-paths calculated for an intermediate node v. Thus, even though it calculates at most two s-t-paths, the BDA$_{2\text{SSP}}$ may compute $\ell - 1$ (not ℓ because s-t-paths are not propagated) s-v-paths for an intermediate node v. Using the running time bound of the BDA described in Theorem 4.5, we obtain the following result immediately.

Theorem 14.3. *The BDA$_{2\text{SSP}}$ solves a 2-SSP instance $\mathcal{I} := (D, s, t, c, 2)$ in time*

$$\mathcal{O}\left(n\ell \log n + \ell m\right) \in \mathcal{O}\left(n(n \log n + m)\right). \tag{46}$$

Based on the memory consumption derived for the BDA in Theorem 4.6, we conclude this section stating the space consumption bound of the BDA$_{2\text{SSP}}$.

Theorem 14.4. *The memory usage of the BDA$_{2\text{SSP}}$ is in*

$$\mathcal{O}\left(n\ell + n + m\right) \in \mathcal{O}\left(n^2 + m\right). \tag{47}$$

Proof. The BDA uses $\mathcal{O}\left(N + n + m\right)$ space where $N = \sum_{v \in V} N_v$ and N_v is the number of efficient s-v-paths calculated by the algorithm for the node $v \in V$. We have $N \leqslant nN_{\max}$ with $N_{\max} = \max_{v \in V} N_v$ and in our BDA$_{2\text{SSP}}$ scenario $N_{\max} < \ell$ as discussed already. The modifications defined in Definition 14.5 to the original BDA to obtain the BDA$_{2\text{SSP}}$ do not have any further impact on the space consumption. □

14.3.3 Second Shortest Simple Paths are Shortest Paths

In this section we prove that a second shortest simple path is a shortest simple path in a modified digraph. This is important in the development of the k-SSP algorithm introduced in Section 14.4. Whenever we remove a path p from a given digraph D, we write $D \setminus p$ and we delete the nodes (and consequently the arcs) of p from D.

It is easy to see as it follows directly from Definition 14.3 that in a solution sequence $P = (p_1, \dots, p_k)$ to a k-SSP instance, p_1 is the parent path of p_2. Then, we can formulate the following important lemma.

Lemma 14.2. *Consider a 2-SSP instance $\mathcal{I} := (D, s, t, c, 2)$ and let $P = (p_1, p_2)$ be a solution sequence. Assume that (v, w) is p_2's deviation arc. The suffix path $p_2^{w \to t}$ is a shortest w-t-path in the digraph $D \setminus p_2^{s \to v}$.*

Proof. We can write $p_2 = p_1^{s \to v} \circ (v, w) \circ p_2^{w \to t}$. If a w-t-path $q^{w \to t}$ with $c(q^{w \to t}) < c(p_2^{w \to t})$ exists in $D \setminus p_2^{s \to v}$ we have $c(p_2) > c(p_1^{s \to v} \circ (v, w) \circ q^{w \to t})$ and p_2 would not be a second shortest s-t-path in D. □

As a consequence, we formulate the following result.

Lemma 14.3. *A second simple shortest path is given by*

$$\begin{aligned} p_2 := \arg\min \Big\{ c(q) \;\Big|\; & q = p_1^{s\to v} \circ (v,w) \circ q^{w\to t}, \\ & q^{w\to t} \textit{ shortest } w\text{-}t\textit{-path in } D \setminus p_1^{s\to v}, \\ & (v,w) \in \delta^+(v) \setminus p_1,\, v \in p_1 \Big\}. \end{aligned} \tag{48}$$

We observe that the solution p_2 in (48) is contained in the solution set (43) of a Replacement Path (RP) instance. Thus, in a worst case scenario, the BDA$_{2SSP}$ needs to do as much effort as an RP algorithm to find p_2 in the set (43) of RP solutions. This intuition is formally mirrored in Williams and Williams (2018, Theorem 1.1). The result states that while currently having the same complexity, a truly subcubic 2-SSP algorithm implies a truly subcubic RP algorithm. I.e., it is unlikely to design an algorithm solving 2-SSP instances faster than RP instances in the worst case.

However, for practical purposes the fact that the solution p_2 from (48) is included in the set (43) unveils the strength of the BDA$_{2SSP}$ as a 2-SSP algorithm: stopping after at most two paths reach the target node t reduces the number of iterations in comparison to the need to solve $\mathcal{O}(n)$ One-to-One Shortest Path instances to calculate (43).

14.4 K-SPP ALGORITHM BY RODITTY AND ZWICK USING THE BDA

Roditty and Zwick (2012) discuss a black box algorithm for the k-SSP problem. It is a *black box* algorithm because the authors do not specify how to solve the key subroutine in their algorithm: the computation of a second-shortest simple path. Moreover, they do not implement their algorithm in the paper. In this section we fill the gap using the BDA$_{2SSP}$.

The algorithm presented in (Roditty & Zwick, 2012) performs $2k$ computations of a second shortest simple path to solve a k-SSP instance $\mathcal{I} := (D, s, t, c, k)$. It fills the solution sequence P iteratively. In our exposition we assume that at any stage, the deviation tree T_P associated with P exists implicitly. In particular, this allows us to use the notions from Definition 14.3. We discuss this in detail in Remark 14.2. The pseudocode for our new algorithm is in Algorithm 15. We state the following remark to avoid misunderstandings regarding the source nodes in the original k-SSP instance and in the 2-SSP instances solved as subroutines in our algorithm.

Remark 14.1 (Source nodes in $\mathcal{I}_{BOSP}$ instances)**.** *Given an ℓ^{th} shortest path p_ℓ for some $\ell \in \{1, \dots, k\}$, the corresponding BOSP instance $\mathcal{I}_{BOSP}^{p_\ell}$ is defined as in Definition 14.4 but the source node in $\mathcal{I}_{BOSP}^{p_\ell}$ is not always the actual source node s of our k-SSP instance. For $\mathcal{I}_{BOSP}^{p_\ell}$ the source node is $\mathrm{source}(p_\ell)$. As discussed in the previous section, the ℓ^{th} shortest path p_ℓ in the original graph D is not a shortest s-t-path but its suffix $p_\ell^{\mathrm{source}(p_\ell)\to t}$ is a shortest path in a modified version of D.*

The data structures of the algorithm are the solution sequence P and a priority queue C of s-t-paths sorted according to the paths' costs. Both struc-

Algorithm 15: New algorithm for the k-SSP problem.

```
Input   : k-SSP instance $\mathcal{I} := (D, s, t, c, k)$
Output: Solution sequence $P = (p_1, \ldots, p_k)$ of distinct simple $s$-$t$-paths.
Priority queue of candidate $s$-$t$-paths $C \leftarrow \emptyset$;
$p_1 \leftarrow$ shortest $s$-$t$-path in $D$;
Solution sequence $P \leftarrow (p_1)$;
$p_2 \leftarrow \text{BDA}_{2\text{SSP}}$ solution for $\mathcal{I}^{p_1}_{\text{BOSP}}$;
$(v, w) \leftarrow$ Parent deviation arc of $p_2$;
Add $(v, w)$ to $\text{blocked}(p_1)$;
Insert $p_2$ to $C$;
for $\ell \in \{2, \ldots, k\}$ do
    $p_\ell \leftarrow$ Extract path from $C$ with min. cost;
    Add $p_\ell$ to the solution sequence $P$;
    if $\ell == k$ then break;
    $q^{\text{source}(p_\ell) \to t} \leftarrow \text{BDA}_{2\text{SSP}}$ solution for $\mathcal{I}^{p_\ell}_{\text{BOSP}} = (\overline{D}, \text{source}(p_\ell), t, \gamma_{p_\ell})$
     with $\overline{D} = D \setminus p_\ell^{s \to \text{dev}(p_\ell)}$;
    if $q^{\text{source}(p_\ell) \to t} \neq$ NULL then
        New $s$-$t$-path $q \leftarrow p_\ell^{s \to \text{source}(p_\ell)} \circ q^{\text{source}(p_\ell) \to t}$;
        $(v', w') \leftarrow$ Parent deviation arc of $q$;
        Add $(v', w')$ to $\text{blocked}(p_\ell)$;
        Insert $q$ into $C$;
    $p \leftarrow$ Parent path of $p_\ell$;
    $q^{\text{source}(p) \to t} \leftarrow \text{BDA}_{2\text{SSP}}$ solution for $\mathcal{I}^{p}_{\text{BOSP}} = (\overline{D}, \text{source}(p), t, \gamma_p)$ with
     $\overline{D} = D \setminus (p^{s \to \text{dev}(p)} \cup \text{blocked}(p))$;
    if $q^{\text{source}(p) \to t} \neq$ NULL then
        New $s$-$t$-path $q \leftarrow p^{s \to \text{source}(p)} \circ q^{\text{source}(p) \to t}$;
        $(v', w') \leftarrow$ deviation arc of $q$;
        Add $(v', w')$ to $\text{blocked}(p)$;
        Insert $q$ into $C$;
return $P$;
```

tures are initially empty. In its initialization phase, the algorithm computes a shortest s-t-path p_1 in D w.r.t c and stores it in P as the first solution in the solution sequence (Line 2 and Line 3). Additionally, a second shortest path p_2 is computed applying the $\text{BDA}_{2\text{SSP}}$ to the $\mathcal{I}^{p_1}_{\text{BOSP}}$ instance (Line 4). The obtained path is inserted into C (Line 7). p_1 is the parent path of p_2.

Every path in P has a list of blocked arcs associated with it. For a path p, the list $\text{blocked}(p)$ contains the deviation arcs from p's children paths that are already computed. When looking for further deviations from p, we delete the arcs in $\text{blocked}(p)$ from the digraph to ensure that the deviation arcs leading to the already computed children paths of p are not computed again. Thus, since p_2 is a child path of p_1, the deviation arc of p_2 is added to $\text{blocked}(p_1)$ (Line 6).

After the initialization, the main loop of the algorithm starts. If performs $k-1$ iterations. Every iteration $\ell \in [2, k]$ starts with the extraction of a minimal path from C (Line 9) which we call p_ℓ. p_ℓ is immediately added to P after its extraction and it becomes part of the final solution sequence (Line 10). Then, two instances of the 2-SSP are defined and solved.

FIRST 2-SSP CALCULATION Let $(v, w) = (\text{dev}(p_\ell), \text{source}(p_\ell))$ be the deviation arc from p_ℓ as defined in Definition 14.3. Then, we build the instance $\mathcal{I}^{p_\ell}_{\text{BOSP}} = (\overline{D}, w, t, \gamma_{p_\ell})$ with $\overline{D} = D \setminus p_\ell^{s\to v}$. Recall that by Lemma 14.2, the suffix $p_\ell^{w\to t}$ is a shortest w-t-path in $\overline{D}$. Using the $\text{BDA}_{2\text{SSP}}$, a second shortest w-t-path $q^{w\to t}$ in $\overline{D}$ w.r.t. γ_{p_ℓ} is searched. The result if it exists, is a new suffix for the prefix $p_\ell^{s\to w}$. Together, both subpaths build a new candidate s-t-path $q := p_\ell^{s\to w} \circ q^{w\to t}$.

POSTPROCESSING If q is successfully built, p_ℓ is its parent path (see Remark 14.2). Moreover, q's deviation arc is added to the list $\text{blocked}(p_\ell)$. Finally, q is added to C.

SECOND 2-SSP CALCULATION The second $\text{BDA}_{2\text{SSP}}$ query (Line 19) in every iteration looks for the next-cheapest deviation from the parent path p of the extracted path p_ℓ. When building the corresponding 2-SSP instance $\mathcal{I}^{p}_{\text{BOSP}}$, the deviation arcs from p's children paths must be deleted from the digraph D. Otherwise, the solution to $\mathcal{I}^{p}_{\text{BOSP}}$ would be an already computed deviation. Apart from deleting the arcs in $\text{blocked}(p)$ from D, we again delete p's prefix $p^{s\to\text{dev}(p)}$ from p. This ensures that after the $\text{BDA}_{2\text{SSP}}$ computation, the concatenation of $p^{s\to\text{source}(p)}$, where w is the adjacent node to v in p, and the result $q^{w\to t}$ is a simple path. If q is successfully built, the algorithm repeats the postprocessing of the first $\text{BDA}_{2\text{SSP}}$ computation. This query searches for the cheapest simple path alternative for $p^{w\to t}$ without considering the alternatives that have already been computed.

14.4.1 Correctness and Complexity

In this subsection we sketch the correctness proof and the complexity of Algorithm 15. The correctness of Algorithm 15 using a black box algorithm to solve the arising 2-SSP instances is discussed in Roditty and Zwick (2012).

In Algorithm 15 we use the parent-child relationship of paths introduced in Definition 14.3. Formally we would need a proof to show that indeed the computed paths and our usage of this notion in the algorithm are in accordance with the original definition. The following remark gives a strong intuition. The proof can then easily be concluded with an induction step.

Remark 14.2. *As mentioned earlier p_2's parent path is p_1. In the first iteration of Algorithm 15, we thus build a $\mathrm{source}(p_2)$-t-path $q^{\mathrm{source}(p_2)\to t}$ in Line 12. It is used to build an s-t-path $q := p_2^{s\to\mathrm{source}(p_2)} \circ q^{\mathrm{source}(p_2)\to t}$. Since by definition $\mathrm{source}(p_2) \in p_2$ and $\mathrm{source}(p_2) \notin p_1$, q shares a prefix of maximum length with p_2. Hence, p_2 induced the BOSP instance $\mathcal{I}^{p_2}_{BOSP}$ that led to q's computation and p_2 is q's parent path.*

In the second BDA$_{2SSP}$ query in the first iteration, an s-t-path q is computed in Line 19. q already starts at s because $s = \mathrm{source}(p_1)$ and thus s is the source node in $\mathcal{I}^{p_1}_{BOSP}$ (cf. Remark 14.1). In the corresponding digraph, the arcs in $\mathrm{blocked}(p_1)$ are deleted. At this stage, the list only contains p_2's deviation arc $(v, w) = (\mathrm{dev}(p_2), \mathrm{source}(p_2))$ that was added to the list in Line 6. Hence if $q^{s\to\mathrm{dev}(q)}$ coincides with p_1 until a node $\mathrm{dev}(q)$ that comes after $\mathrm{dev}(p_2)$ along p_1, q shares a longest prefix with p_1. Otherwise, if $\mathrm{dev}(q)$ does not come after $\mathrm{dev}(p_2)$ along p_1, q shares a longest prefix with p_1 and p_2. By definition, the parent path of q is then set to be the first of these paths in P, i.e., p_1.

Recall that a child's deviation node does not come before its parent's deviation node as remarked already in Section 14.2. Then, we repeat the arguments from the last paragraphs for any path extracted from C in Line 9 of Algorithm 15 to prove that the notions from Definition 14.3 are correctly used in Algorithm 15.

Algorithm 15 deletes prefixes from D to ensure that it can generate distinct and simple paths when concatenating the suffixes built by the BDA$_{2SSP}$ with the deleted prefix in the corresponding parent path (see Lemma 14.5).

Lemma 14.4. *Let p be an s-t-path in the solution sequence P of Algorithm 15 with deviation node v and deviation arc (v, w). Let $q^{w\to t}$ be a second shortest path computed in Line 12 or in Line 19. The s-t-path $q = p^{s\to w} \circ q^{w\to t}$ is simple.*

Proof. For the computation of $q^{w\to t}$, we delete the prefix $p^{s\to v}$ from D to build $\overline{D}$. Hence, both subpaths are node-disjoint. As discussed already, the non-negativity of every γ ensures that the path output by the BDA$_{2SSP}$ is simple. Hence, q does not contain a cycle. □

The deletion of prefixes and blocked arcs in the graphs $\overline{D}$ in Line 12 and in Line 19 ensures that every s-t-path found in Line 14 or in Line 21 of Algorithm 15 is built and added to C only once. This is a property of the deviation tree that partitions the set of s-t-paths into disjoint sets.

Lemma 14.5 (Roditty and Zwick (2012), Lemma 3.3.). *Every s-t-path added to C is only added once.*

Finally, the correctness of Algorithm 15 is proven by induction and using the correctness of the BDA$_{2SSP}$, Lemma 14.4, and Lemma 14.5.

Theorem 14.5 (Roditty and Zwick (2012), Lemma 3.4.). *Algorithm 15 solves the k-SSP problem.*

We end this section analyzing the running time bound and the memory consumption of Algorithm 15.

Theorem 14.6. *Algorithm 15 solves a* k*-SSP instance* $\mathcal{I} := (D, s, t, c, k)$ *in time*

$$\mathcal{O}\left(kn(n\log n + m)\right). \tag{49}$$

Proof. The main loop of Algorithm 15 does $k-1$ iterations. Except in the last iteration where it does not compute new paths, it performs two BDA$_{2SSP}$ computations per iteration. Thus, it computes $(2k-4) \in \mathcal{O}(k)$ new paths using the BDA$_{2SSP}$. Using the running time bound for the BDA$_{2SSP}$ derived in Theorem 14.3, we obtain the running time bound (49) for Algorithm 15. Thereby we can neglect the effort for the concatenation of paths in Line 14 and in Line 21 since they are done in $\mathcal{O}(n)$ given that simple paths have at most $(n-1)$ arcs. Moreover, the priority queue operations on C can also be neglected since the queue contains $\mathcal{O}(k)$ elements and we can assume input values of k s.t. $\mathcal{O}(k\log k) \subset \mathcal{O}(km)$. □

Theorem 14.7. *Algorithm 15 uses* $\mathcal{O}\left(kn + n^2 + m\right)$ *memory.*

Proof. By Theorem 14.4 we know that any BDA$_{2SSP}$ query in Line 12 or in Line 19 requires $\mathcal{O}\left(n^2+m\right)$ space. Algorithm 15 does not run multiple BDA$_{2SSP}$ queries simultaneously. We store the paths in the solution sequence P, using the deviation tree T_P of P (cf Definition 14.3) that allows us to use the parent-child relationships between paths and the notion of deviation arcs, deviation nodes, and source nodes. In T_P every node $v \in V$ can appear multiple times, one per path in P. Since simple paths have at most $n-1$ arcs, this results in $\mathcal{O}(kn)$ space. □

14.4.2 Implementation details

The performance of Algorithm 15 in practice depends on the number of iterations required by the BDA$_{2SSP}$ queries. Intuitively, we hope that the search deviates from and returns to the path input to the algorithm after only a few iterations. On big graphs, finding k simple paths between the input nodes s and t is most often a *local* search since only a relatively small number of nodes needs to be explored. However, a BDA$_{2SSP}$ query that deviates from the input path but does not return to it fast resembles a One-to-All BOSP algorithm. Thus, it performs a rather *global* search that requires a lot of time.

The behavior defined above happens mainly when the target node t is not reachable from the source node $\text{source}(p)$ of a BDA$_{2SSP}$ query defined based on an s-t-path p. More precisely, there is always a $\text{source}(p)$-t-path in the considered digraph, namely the subpath $p^{\text{source}(p)\to t}$ but we are interested in a second shortest $\text{source}(p)$-t-path. However if p's suffix is the only path, the BDA$_{2SSP}$ terminates when its priority queue is empty at the beginning of an iteration. This behavior motivates the following *pruning technique*.

BDA$_{2SSP}$ PRUNING USING THE PATHS' QUEUE As soon as Algorithm 15 has at least k s-t-paths in P and in C, i.e., as soon as $|P| + |C| \geqslant k$, we can possibly end $\mathrm{BDA}_{2\mathrm{SSP}}$ queries before t is reached or before the queue of the $\mathrm{BDA}_{2\mathrm{SSP}}$ is emptied. In this scenario, we set $\bar{c} := \max_{p \in C} c(p)$. If the $\mathrm{BDA}_{2\mathrm{SSP}}$ extracts a path q with $c(q) \geqslant \bar{c}$, the lexicographic ordering of the extracted paths during the $\mathrm{BDA}_{2\mathrm{SSP}}$ guarantees that no s-t-path p with costs $c(p) < \bar{c}$ can be build using the suffix computed in that query. Hence, the $\mathrm{BDA}_{2\mathrm{SSP}}$ query can be aborted. Note that the condition $|P| + |C| \geqslant k$ is met after the $\frac{k}{2}^{\text{th}}$ iteration at the earliest because in every iteration Algorithm 15 generates at most 2 new s-t-paths.

PRUNING BY MIN PATHS' QUEUE COSTS In graphs with multiple cost-equivalent s-t-paths we may avoid some $\mathrm{BDA}_{2\mathrm{SSP}}$ queries. Suppose c^* is the minimum cost of paths stored in C at the beginning of an iteration, i.e., $c^* = \min_{p \in C}\{c(p)\}$. We denote the set of paths in C with costs c^* by $C^* \subseteq C$. If at the beginning of an iteration in Algorithm 15 we have $|P| + |C^*| \geqslant k$, we can terminate the algorithm after extracting the first $k - |P|$ paths from the priority queue and storing them in P. The avoided $\mathrm{BDA}_{2\mathrm{SSP}}$ queries would yield s-t-paths p with $c(p) \geqslant c^*$ and thus would not destroy the optimality of the output sequence P.

14.5 EXPERIMENTS

In this section we assess the practical performance of Algorithm 15 by comparing it to the current state of the art k-SSP algorithm: the KM algorithm introduced in Kurz and Mutzel (2016).

14.5.1 Benchmark Setup

We benchmark Algorithm 15 on 100×100 grid graphs and on road networks from parts of the USA. Both types of graphs have already been used previous chapters of the thesis (Section 6.2.1). Instead of considering multiple arc cost components, we use the first one only to define the scalar arc costs c needed in k-SSP instances.

GRID GRAPHS On the digraph D that has 10000 nodes and 39600 arcs, we define 10 different scalar arc cost functions c. The arc costs are chosen uniformly and at random between 0 and 10. Each of these cost functions, paired with the grid graph, builds a pair (D, c). For each of these pairs, we define 200 s-t-pairs, where s and t are chosen uniformly at random from the set of nodes in D. Finally, for every tuple (D, s, t, c), we define a k-SSP instance $\mathcal{I} := (D, s, t, c, k)$ using different values for k as shown in Table 22.

ROAD NETWORKS Recall that the size of the road networks is shown in Table 2. The arc costs c corresponds to the distance between the arcs' end nodes. We draw 200 s-t-pairs uniformly and at random from each graph's nodes' set. The final k-SSP instances are then defined using different values for k for every tuple (D, s, t, c) as shown in Table 23.

Columns	Unit	Accuracy
Time	s	Hundreds
Speedup = T_{KM}/T_{NA}	\	Hundreds
Iterations	amount	Integer
Trees and BDA_{2SSP}	amount	Integer

Table 20: Details on the report of geometric means in our tables in this section.

BENCHMARK ALGORITHM We compare our implementation of Algorithm 15 that is available in (Maristany de las Casas, 2023d) with the implementation of the KM algorithm (Kurz & Mutzel, 2016) kindly provided to us by the authors. Both algorithms are implemented in C++ and use the same datastructures to store the graph. We explained the choice of the KM algorithm for our benchmarks already in Section 14.1.1.

ENVIRONMENT We used a computer with an Intel(R) Xeon(R) Gold 6338 @ 2.00GHz processor and assigned 30GB of RAM and 2h=7200s for each instance. Both algorithms are compiled using the g++ compiler and the -O3 compiler optimization flag. Our code repository (Maristany de las Casas, 2023d) includes the scripts used to run the KM algorithm (even though the code itself needs to be requested from the authors). The scripts to run it are relevant since the implementation includes some optional arguments that highly impact its performance. Our configuration resembles the performance of the best version of the algorithm in Kurz and Mutzel (2016).

14.5.2 Results

To mitigate the impact of outliers on the reported averages we always report geometric means in this section. In Table 20 we specify the format of the columns used in the tables in this section. We used the publicly available files results/evaluationGrids.ipynb and results/evaluationRoad.ipynb in (Maristany de las Casas, 2023d) to generate the tables and figures. The results folder in this repository also contains the detailed output for every solved instance. For every row in Table 22 and in Table 23 the evaluation scripts generate scatter plots like the ones in Figure 44 - Figure 47.

Speedups are calculated as the time needed by the KM algorithm divided by the time needed by Algorithm 15. Thus, speedups greater than 1 indicate a faster running time for Algorithm 15. Instances that were not solved by any of the two algorithms are not included in our reports. If an instance was solved by one of the algorithms only, we assume a running time of $T = 2h = 7200s$ for the other algorithm.

14.5.3 Grid Graphs

We summarize our results on grid graphs in Table 22. Table 21 explains the column names that are not self-explanatory. For the chosen k-SSP instances on grids, we end up considering 2000 instances for every fixed value of k. As shown in Table 22 the KM algorithm and Algorithm 15 solved all instances with values of k up to 10^5. For $k = 5 \times 10^5$ the KM algorithm fails to solve 49 instances and for $k = 10^6$ it does not solve 445 instances. The KM algorithm fails to solve all these instances because it hits the memory limit. In contrast, Algorithm 15 manages to solve all instances for every value of k. Regarding the speedup, we observe that Algorithm 15 consistently outperforms the KM algorithm. Moreover, the speedup increases as k increases. For $k \geqslant 5 \times 10^4$ the speedup is close to or higher than an order of magnitude.

The reason for both the unsolved instances and the slower running times of the KM algorithm is that the algorithm is forced to compute too many shortest path trees as shown in the column *Trees* in Table 22. In fact, for $k \geqslant 10^4$ it approximately needs to compute a tree for every 5th solution path. This is because the considered grid graph is originally an undirected graph. After converting it to a directed graph by adding antiparallel arcs, it contains many cycles. The KM algorithm initially computes a shortest path tree and it can build $\mathcal{O}(m)$ paths from that tree by switching tree arcs and non-tree arcs. This procedure works as long as the switch does not cause the *next* s-t-path to be non-simple. Given the great amount of antiparallel arcs in the considered grid graph, the KM algorithm cannot build many simple paths from one shortest path tree.

The good performance of Algorithm 15 on grid graphs is due to the low number of iterations that it requires in every BDA$_{2SSP}$ search. The column *BDA$_{2SSP}$* in Table 22 reports how many out of at most $2k$ BDA$_{2SSP}$ queries are performed on average. We see that due to the *Pruning by Min Paths' Queue Costs* described in Section 14.4.2, Algorithm 15 can skip around 20% of the BDA$_{2SSP}$ queries on average. Whenever the conducted queries find a new path, the average number of iterations in every BDA$_{2SSP}$ query ranges from 20 to 28 as shown in the column *Iterations* ✓. This means that the computed second simple shortest paths in Line 12 and in Line 19 are found fast. Moreover, in column *BDA$_{2SSP}$* ✗ we report the number of BDA$_{2SSP}$ queries that do not find a suitable suffix to build a new s-t-path. On these searches the BDA$_{2SSP}$ queries can fail either because t is not reachable or because the *BDA$_{2SSP}$ Pruning using the Paths' Queue* explained in Section 14.4.2 avoids the computation of the suffix. As reported in the column *Iterations* ✗, the stopping condition is fulfilled after 16 to 17 BDA$_{2SSP}$ iterations hence avoiding the computation of unneeded and large sets of efficient paths.

Algorithm	Column Name	Explanation
KM	Trees	Shortest path trees computed on average.
Algorithm 15	BDA_{2SSP}	BDA_{2SSP} queries on average. At most $2k-4$.
	BDA_{2SSP} ✗	Average number of BDA_{2SSP} queries that did not reach t.
	Iterations ✓	Avg. iterations in BDA_{2SSP} queries that reached t.
	Iterations ✗	Avg. iterations in BDA_{2SSP} queries that did not reach t.

Table 21: Explanation of columns in Table 22 and Table 23.

Table 22: Summarized results obtained from the k-SSP instances defined on grid graphs. The column *BDA_{2SSP}* ✗ reports the number of BDA_{2SSP} runs that did not return a second shortest path. If *Iterations* ✗ is a small number, the non-existence of a relevant second shortest path could be proven fast (cf. Section 14.4.2).

k	KM			Algorithm 15						SPEEDUP
	Solved	Trees	Time	Solved	BDA_{2SSP}	BDA_{2SSP} ✗	Iterations ✓	Iterations ✗	Time	
1000	2000	97	0.05	2000	1604	245	28	17	0.01	3.68
5000	2000	778	0.25	2000	8182	1303	25	17	0.05	4.66
10000	2000	1847	0.53	2000	16387	2656	25	16	0.10	5.17
50000	2000	11207	4.82	2000	82481	13541	23	16	0.49	9.76
100000	2000	24011	9.95	2000	167223	28760	22	16	1.03	9.70
500000	1951	130124	65.29	2000	836001	142677	21	16	6.16	10.61
1000000	1555	220368	249.81	2000	1681693	291944	20	16	12.19	20.50

14.5.4 Road Networks

In Table 23 we summarize the results obtained on road networks. For every road network and every value of k the table contains a row showing the average results over the 200 possibly solved instances in that group.

SOLVABILITY The first noticeable difference between both algorithm is that even on the smallest NY network, the KM algorithm fails to solve a considerable amount of instances when $k \geqslant 5 \times 10^4$ (see also Figure 44). Interestingly, also on the much bigger networks, $k = 5 \times 10^4$ constitutes a threshold beyond which the KM algorithm struggles to solve multiple instances. Figure 45 shows an example. A look at the *KM Time* column in Table 23 unveils that the average running times of the KM algorithm are way below the time limit $T = 7200$s. Indeed, the KM algorithm's bottleneck regarding solvability is, as on grid graphs, the memory limit of 30GB. On graphs smaller than LKS, Algorithm 15 manages to solve $\geqslant 90\,\%$ of the instances with $k < 10^6$. The percentage of solved instances with $k \geqslant 5 \times 10^5$ on the LKS and the CTR networks decreases rapidly. Whenever Algorithm 15 fails to solve an instance, its because it hits the memory limit. It is worth noting that instances with $k \geqslant 10^5$ on road networks are novel in the k-SSP literature for algorithms matching the running time bound derived in Theorem 14.6.

RUNNING TIMES For every value of $k \geqslant 10$ and for every considered road network, Algorithm 15 is faster than the KM algorithm on average. In Table 23 we can observe that the speedup is proportional to the value of k. The actual values for the speedup favor Algorithm 15 most clearly on the BAY instances. On this graph, speedups of over an order of magnitude on average are reached for $k = 5000$ already. The speedup correlates with the number of shortest path trees required by the KM algorithm. The BAY network is particularly hard in this regard.

There are multiple instances defined on the FLA and LKS networks for which k s-t-paths with the cost of a shortest path exist. On these networks, the KM algorithm solves instances computing less than 51 and less than 10 shortest path trees, respectively. For this reason, the speedup achieved by Algorithm 15 on these graphs is smaller (cf. Figure 46). Here, the running time of the KM algorithm is dominated by the checks to determine if computed paths are simple. Despite the enhanced performance of the KM algorithm on these instances, Algorithm 15 outperforms the KM algorithm for large values of k w.r.t. solvability and speed (cf. Figure 47).

The pruning techniques discussed in Section 14.4.2 work well in practice (see Table 23). The column *BDA*$_{2SSP}$ ✗ reports the average number of BDA$_{2SSP}$ queries that do not find a relevant second shortest simple path. These searches, as explained in Section 14.4.2, could cause the BDA$_{2SSP}$ queries to compute minimal complete sets of efficient paths for every reachable node. However, using our pruning techniques, we can see in the column *Iterations* ✗ that the average number of iterations on these searches remains low. Often the required iterations on average are even lower than the iterations needed in successful BDA$_{2SSP}$ queries (see column *Iterations* ✓).

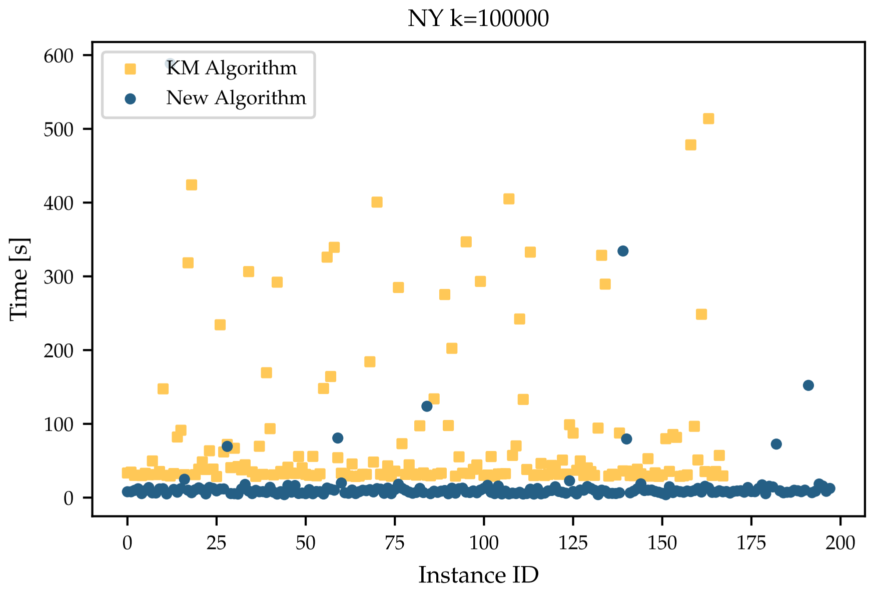


Figure 44: Results obtained from the 200 instances (if solved) defined on NY networks with $k = 10^5$.

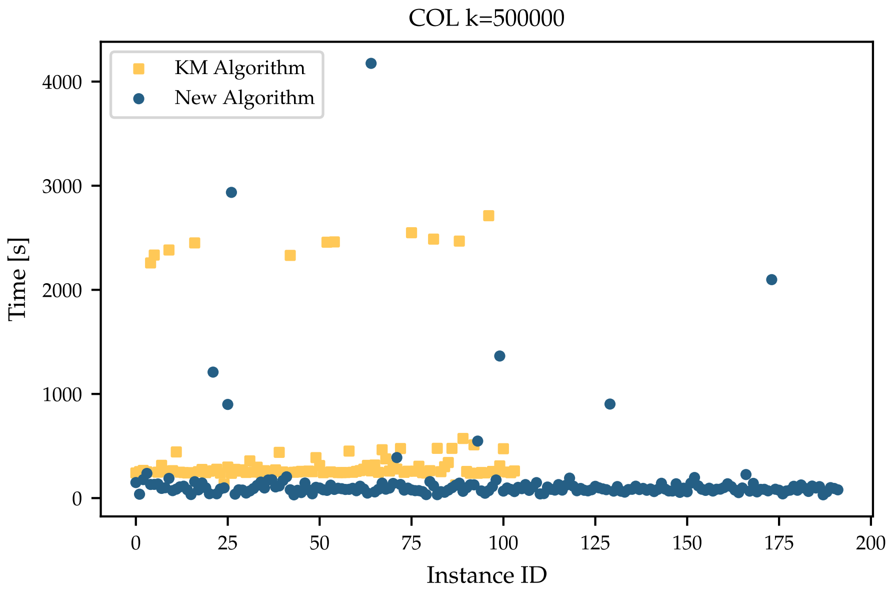


Figure 45: Results obtained from the 200 instances (if solved) defined on COL networks with $k = 5 \times 10^5$.

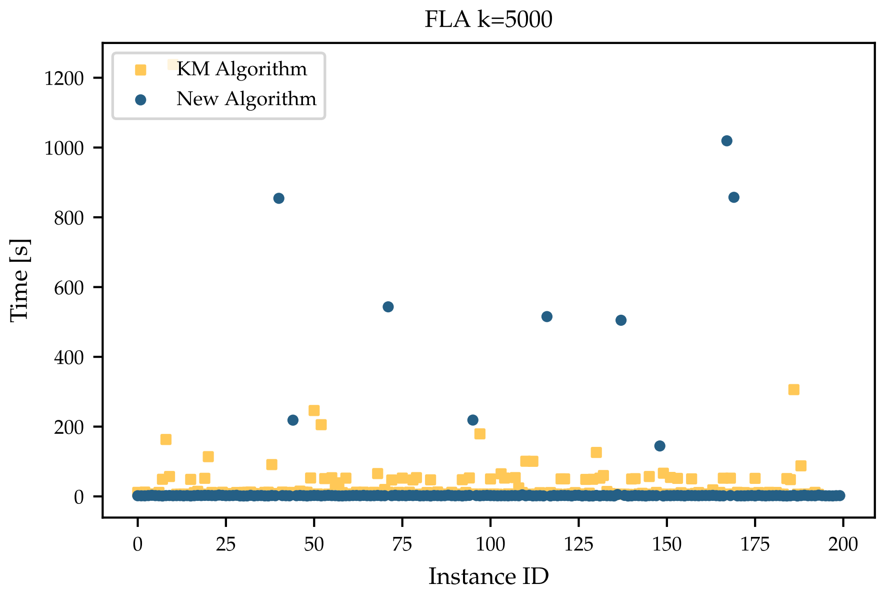


Figure 46: Results obtained from the 200 instances defined on FLA networks with $k = 5000$.

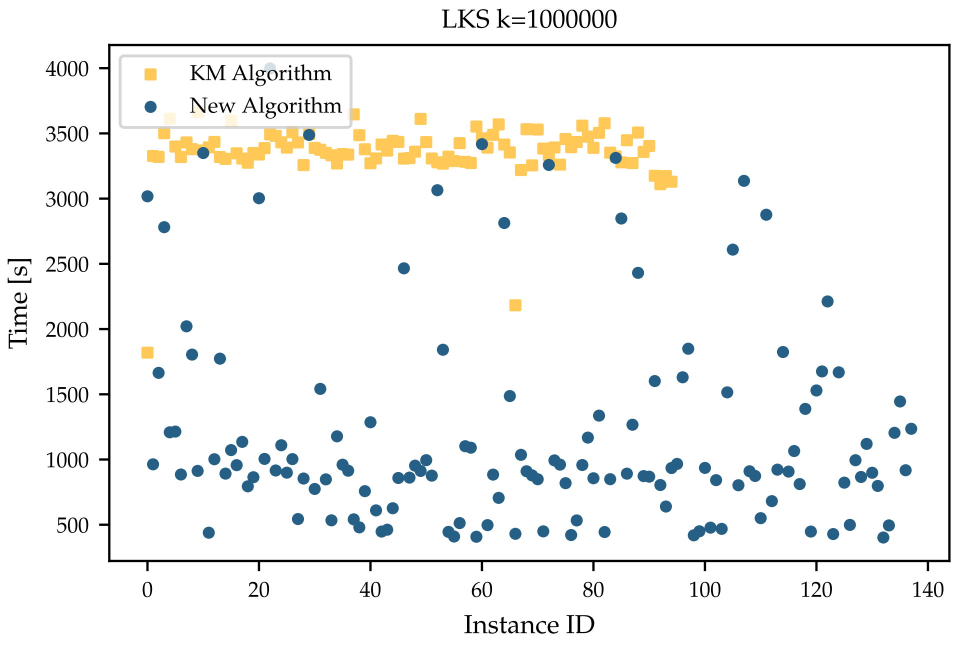


Figure 47: Results obtained from the 200 instances (if solved) defined on LKS networks with $k = 10^6$.

Table 23: k-SSP instances on road networks. If *Iterations* ✗ is a small number, the non-existence of a relevant second shortest path could be proven fast (cf. Section 14.4.2).

k	KM			Algorithm 15						Speedup
	Solved	Trees	Time	Solved	BDA_{2SSP}	BDA_{2SSP} ✗	Iterations ✓	Iterations ✗	Time	
				NY						
10	200	1	0.05	200	15	4	186	130	0.10	0.52
100	200	4	0.15	200	188	51	140	98	0.12	1.28
1000	200	20	1.13	200	1935	523	112	88	0.23	4.98
5000	200	109	4.48	200	9760	2677	100	82	0.68	6.57
10000	197	220	8.98	200	19552	5363	96	79	1.18	7.61
50000	182	987	47.34	200	97824	26736	88	76	5.20	9.11
100000	168	1643	113.67	198	196277	53170	84	69	9.73	11.68
500000	125	4405	867.08	198	980103	266634	78	66	48.72	17.80
1000000	91	4116	1975.73	196	1959033	532676	77	61	90.82	21.75
				BAY						
10	200	2	0.07	200	14	3	215	168	0.10	0.71
100	200	6	0.20	200	189	50	182	144	0.12	1.67
1000	200	67	2.00	200	1961	520	145	134	0.30	6.70
5000	198	439	11.17	200	9868	2650	128	123	0.96	11.63
10000	190	847	23.18	200	19758	5303	122	119	1.70	13.60
50000	151	3598	170.32	199	98883	26579	111	113	7.97	21.37
100000	127	5710	438.19	199	197836	53240	107	113	16.77	26.13
500000	61	12297	2528.40	190	991928	263977	104	81	77.39	32.67

1000000	40	15572	4113.70	189	1983487	526322	101	79	138.60	29.68
					COL					
10	200	1	0.09	200	11	3	128	152	0.13	0.69
100	200	3	0.22	200	141	46	109	133	0.15	1.41
1000	200	11	1.81	200	1557	420	79	123	0.33	5.45
5000	198	38	9.61	200	8300	2506	70	120	1.10	8.76
10000	193	62	16.38	200	16930	4763	70	120	2.09	7.83
50000	156	98	138.73	198	86874	25251	73	112	11.11	12.49
100000	140	117	337.81	195	174875	48851	73	97	23.33	14.48
500000	104	115	1433.21	192	883570	234159	69	92	106.66	13.44
1000000	92	124	2396.19	188	1802480	459770	72	86	202.93	11.81
					CAL					
10	200	1	0.46	200	11	4	258	281	0.58	0.79
100	200	2	0.83	200	141	45	257	203	0.63	1.32
1000	200	5	4.20	200	1680	374	223	193	1.24	3.38
5000	197	16	19.48	200	9122	2374	214	186	4.43	4.39
10000	192	25	42.71	200	18463	4760	204	178	8.56	4.99
50000	166	61	317.06	197	95921	24492	185	145	47.17	6.72
100000	152	81	654.53	198	192603	49266	179	149	92.33	7.09
500000	93	208	2942.28	195	964115	246741	165	129	454.34	6.48
1000000	82	325	4287.38	178	1946894	500079	172	124	927.41	4.62
					FLA					
10	200	1	0.21	200	10	4	176	162	0.32	0.66

100	200	2	0.52	200	132	41	147	194	0.37	1.41
1000	200	5	3.46	200	1468	417	105	198	0.80	4.34
5000	194	14	17.44	200	7853	2448	78	181	2.63	6.64
10000	188	22	30.58	200	15763	4858	72	179	5.17	5.91
50000	168	51	184.12	193	80003	23883	67	113	24.91	7.39
100000	147	43	505.38	192	161864	47771	72	107	42.20	11.98
500000	114	38	2190.95	187	821343	229620	85	98	220.34	9.94
1000000	92	51	3413.61	174	1699749	464313	111	88	512.08	6.67
LKS										
10	200	1	0.53	200	10	3	172	242	0.82	0.64
100	200	1	0.99	200	114	45	175	176	0.87	1.14
1000	200	2	5.25	200	1242	344	139	138	1.52	3.46
5000	198	3	26.58	200	6638	2102	130	167	4.71	5.65
10000	194	4	52.56	200	13596	4470	123	176	9.25	5.68
50000	182	4	317.58	199	69638	23313	102	144	49.39	6.43
100000	172	4	653.23	199	141236	47241	97	144	96.96	6.74
500000	122	5	2883.55	171	735793	231196	95	134	657.57	4.39
1000000	95	7	4692.76	138	1595516	437025	97	131	1465.32	3.20
CTR										
10	200	1	4.39	200	9	4	200	285	5.42	0.81
100	200	1	8.63	200	105	43	194	204	5.70	1.51
1000	200	1	45.66	200	1086	288	159	153	9.42	4.85
5000	199	1	213.61	199	5585	1943	142	147	32.08	6.66
10000	198	1	431.95	199	11366	4135	129	150	69.49	6.22

50000	168	2	2135.44	199	59046	22443	107	162	433.13	4.93
100000	154	2	4034.49	198	119179	43441	109	151	916.97	4.40

14.6 CONCLUSION

We use the black box k-Shortest Simple Path (k-SSP) algorithm by Roditty and Zwick (2012) as a baseline to design a new k-SSP algorithm. This algorithm solves at most $2k$ instances of the Second Shortest Simple Path (2-SSP) problem as a subroutine. In their original paper, Roditty and Zwick do not specify how to solve the subroutine efficiently. Since it is a scalar optimization problem, solving it using biobjective path search sounds counter intuitive. However, in this chapter we have shown that the 2-SSP problem can be solved as a Biobjective Shortest Path problem. Given the improved running time bound achieved by the Biobjective Dijkstra Algorithm (Theorem 4.5), our k-SSP algorithm has the same asymptotic running time bound as state of the art k-SSP algorithms. Moreover, the newly defined bidimensional arc cost function allows us to avoid the nodewise comparison of paths to determine if the computed (sub)paths are simple. A constant time dominance check suffices. Given a shortest path with ℓ nodes, other k-SSP algorithms need ℓ One-to-One Shortest Path computations to find a second shortest path. Our biobjective approach considers these ℓ searches in one biobjective path search and stops as soon as the required second shortest path is found. For these reasons we are able to solve large scale k-SSP instances efficiently in practice. Our experiments support this claim.

Part V

CONCLUSION AND BIBLIOGRAPHY

We introduced the *Multiobjective Dijkstra Algorithm* (MDA), a label-setting Multiobjective Shortest Path (MOSP) algorithm with, to the best of our knowledge, the lowest output sensitive running time bound known so far in the literature. On a big set of instances, we could prove the algorithm to also outperform existing MOSP algorithms in practice. This motivated us to design variants of the MDA for different problem variants of MOSP.

For the One-to-One MOSP problem, the *Targeted Multiobjective Dijkstra Algorithm* (T-MDA) combines A^* techniques with a new *pseudo-lazy* management of explored paths. Lazy management of explored paths is nowadays common practice in the implementation of efficient algorithms for the single-criterion Shortest Path problem. When used, explored paths are all stored in the algorithm's priority queue to avoid decrease key operations. In the biobjective scenario the same technique was proven to work well. However, for higher-dimensional arc costs the size of the queue can grow too fast causing an efficiency loss of the algorithm at hand. In our new pseudo-lazy management, we keep the size of the priority queue bounded and index the explored paths that are not in the queue according to their last arc. This indexing allows us to hold back paths from the queue while keeping them sorted in their lists of explored paths using only constant time insertions into the lists. By doing so the T-MDA has the same asymptotic running time bound as the MDA but is notably faster in practice. Other One-to-One MOSP algorithms from the literature turn out to be also slower than the T-MDA in our experiments.

Variants of the MOSP problem with arc cost functions that depend on the paths traversed before reaching the arcs' tail node generalize the Time-Dependent Shortest Path problem. This generalization is not well studied in the literature but relevant for the industry cooperation with Lufthansa Systems GmbH that motivated this thesis. We thus studied the possibilities and limitations of this model. In a multiobjective scenario, these so called *state-dependent* arc cost functions can lead to very dense solution sets. It thus makes sense to study approximation algorithms for MOSP problem variants. The *Multiobjective Dijkstra FPTAS* is a new FPTAS for MOSP and the first one for MOSP problems with state-dependent arc cost functions.

The third part of the thesis is devoted to the use of the MDA as a subroutine to solve other problems than the MOSP problem. The *Implicit Graph MDA* (IG-MDA) can be used to solve discrete Multiobjective Dynamic Programming problems like the *Multiobjective Minimum Spanning Tree* (MO-MST) problem that we studied in the thesis. While doing so, we combined new and known pruning criteria with the new algorithm to solve larger instances than before in the literature. Also the running times on three- and four-dimensional MO-MST instances from the literature were better when the IG-MDA was used to solve them. Finally, we used the biobjective version of the MDA, the BDA, to solve the *Second-Shortest Simple Path* problem. This problem is the main subroutine in a novel algorithm for the k-*Shortest Simple Path* algorithm by Roditty and Zwick (2012). However, the authors did not specify how to solve the subroutine. We closed this gap using a variant of the BDA and thus obtained the first implementation of the algorithm by Roditty and Zwick (2012). Even using Biobjective Optimization, it has a state of the art asymptotic running time bound and in practice it outperforms a previous algorithm from the literature.

All in all, the content of this thesis hopefully makes the modeling of Shortest Path problems using multiple objectives more appealing in theory and in practice. All codes and results are publicly available.

BIBLIOGRAPHY

Maristany de las Casas, P., Sedeño-Noda, A., & Borndörfer, R. (2021). An Improved Multiobjective Shortest Path Algorithm. *Computers and Operations Research*, *135*, 105424. https://doi.org/10.1016/j.cor.2021.105424

Maristany de las Casas, P., Borndörfer, R., Kraus, L., & Sedeño-Noda, A. (2021). An FPTAS for Dynamic Multiobjective Shortest Path Problems. *Algorithms*, *14*(2). https://doi.org/10.3390/a14020043

Maristany de las Casas, P., Kraus, L., Sedeño-Noda, A., & Borndörfer, R. (2023). Targeted multiobjective Dijkstra algorithm. *Networks*. https://doi.org/10.1002/net.22174

Maristany de las Casas, P., Sedeño-Noda, A., & Borndörfer, R. (2023). New Dynamic Programming Algorithm for the Multiobjective Minimum Spanning Tree Problem. https://doi.org/10.48550/ARXIV.2306.16203

Maristany de las Casas, P., Sedeño-Noda, A., Borndörfer, R., & Huneshagen, M. (2023). K-Shortest Simple Paths Using Biobjective Path Search. https://doi.org/10.48550/ARXIV.2309.10377

Ehrgott, M. (2005). *Multicriteria Optimization*. Springer-Verlag. https://doi.org/10.1007/3-540-27659-9

Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc.

Mehlhorn, K. (1984). *Data Structures and Algorithms 1: Sorting and searching*. Springer Berlin Heidelberg.

Korte, B., & Vygen, J. (2005). *Combinatorial Optimization: Theory and algorithms (algorithms and combinatorics)* (W. J. Cook, B. Korte, L. Lovász, A. Wigderson, & G. M. Ziegler, Eds.; Third, Vol. 21). Springer.

Kraus, L. (2021). A Label Setting Multiobjective Shortest Path FPTAS [Bachelor's thesis, Freie Universität Berlin].

Huneshagen, M. (2023). *On the k-Shortest Simple Paths Problem Using Biobjective Search* (Master's thesis). Freie Universität Berlin.

Schrijver, A. (2012). On the history of the shortest path problem. *Documenta Mathematica, Extra Vol: Optimization Stories*, 155–167. https://www.math.uni-bielefeld.de/documenta/vol-ismp/32_schrijver-alexander-sp.pdf

Geisberger, R., Sanders, P., Schultes, D., & Delling, D. (2008). Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. *Workshop on Engineering Applications*. https://api.semanticscholar.org/CorpusID:777101

Delling, D., Sanders, P., Schultes, D., & Wagner, D. (2009). Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, & K. A. Zweig (Eds.), *Algorithmics of large and complex networks: Design, analysis, and simulation* (pp. 117–139). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-02094-0_7

Delling, D., & Wagner, D. (2009). Time-Dependent Route Planning. In *Robust and online large-scale optimization* (pp. 207–230). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-05465-5_8

Nannicini, G. (2009). *Point-to-point shortest paths on dynamic time-dependent road networks* (Theses). Ecole Polytechnique X. https://pastel.hal.science/pastel-00005275

Bast, H., Delling, D., Goldberg, A., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., & Werneck, R. F. (2015). Route Planning in Transportation Networks.

Blanco, M., Borndörfer, R., Hoang, N.-D., Kaier, A., Schienle, A., Schlechte, T., & Schlobach, S. (2016). Solving Time Dependent Shortest Path Problems on Airway Networks Using Super-Optimal Wind. In M. Goerigk & R. F. Werneck (Eds.), *16th symposium on algorithmic approaches for transportation modelling, optimization, and systems (atmos 2016)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany. https://doi.org/10.4230/OASICS.ATMOS.2016.12

Blanco, M., Borndörfer, R., Hoàng, N. D., Kaier, A., Maristany de las Casas, P., Schlechte, T., & Schlobach, S. (2017). Cost Projection Methods for the Shortest Path Problem with Crossing Costs. In G. D'Angelo & T. Dollevoet (Eds.), *17th symposium on algorithmic approaches for transportation modelling, optimization, and systems (atmos 2017)* (15:1–15:14). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/OASIcs.ATMOS.2017.15

Blanco, M., Borndörfer, R., & Maristany de las Casas, P. (2022). An A* Algorithm for Flight Planning Based on Idealized Vertical Profiles. In M. D'Emidio & N. Lindner (Eds.), *22nd symposium on algorithmic approaches for transportation modelling, optimization, and systems (atmos 2022)* (1:1–1:15). Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/OASIcs.ATMOS.2022.1

Baum, M., Dibbelt, J., Wagner, D., & Zündorf, T. (2020). Modeling and Engineering Constrained Shortest Path Algorithms for Battery Electric Vehicles. *Transportation Science*, *54*(6), 1571–1600. https://doi.org/10.1287/trsc.2020.0981

Euler, R., Lindner, N., & Borndörfer, R. (2022). Price Optimal Routing in Public Transportation. https://doi.org/10.48550/ARXIV.2204.01326

Martins, E. Q. V. (1984). On a multicriteria shortest path problem. *European Journal of Operational Research*, *16*(2), 236–245. https://www.sciencedirect.com/science/article/pii/0377221784900778

Sedeño-Noda, A., & Colebrook, M. (2019). A Biobjective Dijkstra Algorithm. *European Journal of Operational Research*, *276*(1), 106–118. https://doi.org/10.1016/j.ejor.2019.01.007

Hansen, P. (1980). Bicriterion Path Problems. In G. Fandel & T. Gal (Eds.), *Multiple criteria decision making theory and application* (pp. 109–127). Springer Berlin Heidelberg.

Garey, M. R., & Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

Raith, A., & Ehrgott, M. (2009). A comparison of solution strategies for biobjective shortest path problems. *Computers & Operations Research*, *36*(4), 1299–1331. https://doi.org/10.1016/j.cor.2008.02.002

Sanders, P., & Mandow, L. (2013). Parallel Label-Setting Multi-objective Shortest Path Search. *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. https://doi.org/10.1109/ipdps.2013.89

Duque, D., Lozano, L., & Medaglia, A. L. (2015). An exact method for the biobjective shortest path problem for large-scale road networks. *European Journal of Operational Research*, *242*(3), 788–797. https://doi.org/10.1016/j.ejor.2014.11.003

Ulloa, C. H., Yeoh, W., Baier, J. A., Zhang, H., Suazo, L., & Koenig, S. (2020). A Simple and Fast Bi-Objective Search Algorithm. *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, 143–151.

Ahmadi, S., Tack, G., Harabor, D., & Kilby, P. (2021). Bi-Objective Search with Bi-Directional A*. In P. Mutzel, R. Pagh, & G. Herman (Eds.), *29th annual european symposium on algorithms (esa 2021)* (3:1–3:15). Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ESA.2021.3

Maristany de las Casas, P., Kraus, L., Sedeño-Noda, A., & Borndörfer, R. (2021). Targeted Multiobjective Dijkstra Algorithm. https://doi.org/10.48550/ARXIV.2110.10978

White, D. J. (1982). The set of efficient solutions for multiple objective shortest path problems. *Computers and Operations Research*, *9*(2), 101–107. https://doi.org/10.1016/0305-0548(82)90008-9

Loui, R. P. (1983). Optimal paths in graphs with stochastic or multidimensional weights. *Communications of the ACM*, *26*(9), 670–676. https://doi.org/10.1145/358172.358406

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik 1*, *1*(1), 269–271. https://doi.org/10.1007/bf01386390

Guerriero, F., & Musmanno, R. (2001). Label Correcting Methods to Solve Multicriteria Shortest Path Problems. *Journal of Optimization Theory and Applications*, *111*(3), 589–613. https://doi.org/10.1023/a:1012602011914

Paixão, J. M., & Santos, J. L. (2013). Labeling Methods for the General Case of the Multi-objective Shortest Path Problem – A Computational Study. In A. Madureira, C. Reis, & V. Marques (Eds.), *Computational Intelligence and Decision Making* (pp. 489–502). Springer Netherlands. https://doi.org/10.1007/978-94-007-4722-7_46

Pulido, F. J., Mandow, L., & Pérez de la Cruz, J. L. (2015). Dimensionality reduction in multiobjective shortest path search. *Computers & Operations Research*, *64*, 60–70. https://doi.org/10.1016/j.cor.2015.05.007

Bökler, F. (2018). *Output-sensitive complexity of multiobjective combinatorial optimization with an application to the multiobjective shortest path problem* (Doctoral dissertation). Technische Universität Dortmund. Technische Universität Dortmund. https://doi.org/10.17877/DE290R-19130

Demeyer, S., Goedgebeur, J., Audenaert, P., Pickavet, M., & Demeester, P. (2013). Speeding up Martins' algorithm for multiple objective shortest path problems. *4OR*, *11*, 323–348. https://doi.org/10.1007/s10288-013-0232-5

Breugem, T., Dollevoet, T., & van den Heuvel, W. (2017). Analysis of FPTASes for the multi-objective shortest path problem. *Comput. Oper. Res.*, *78*, 44–58.

Möhring, R. H. (1999). Verteilte Verbindungssuche im öffentlichen Personenverkehr Graphentheoretische Modelle und Algorithmen. In *Angewandte mathematik, insbesondere informatik* (pp. 192–220). Vieweg + Teubner Verlag. https://doi.org/10.1007/978-3-322-83092-0_11

Emmerich, M., & Deutz, A. (2018). A tutorial on multiobjective optimization: Fundamentals and evolutionary methods. *Natural Computing*, *17*. https://doi.org/10.1007/s11047-018-9685-y

Cormen, T. H. (2022). Introduction to algorithms (C. E. Leiserson, R. L. Rivest, & C. Stein, Eds.; Fourth edition).

Mote, J., Murthy, I., & Olson, D. L. (1991). A parametric approach to solving bicriterion shortest path problems. *European Journal of Operational Research*, *53*(1), 81–92. https://doi.org/10.1016/0377-2217(91)90094-c

Serafini, P. (1987). Some Considerations about Computational Complexity for Multi Objective Combinatorial Problems. In *Recent advances and historical development of vector optimization* (pp. 222–232). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-46618-2_15

Bökler, F. (2017). The multiobjective shortest path problem is NP-hard, or is it? In *Lecture notes in computer science* (pp. 77–87). Springer International Publishing. https://doi.org/10.1007/978-3-319-54157-0_6

Maristany de las Casas, P. (2023a). *maristanyPedro/multiobjectiveDijkstra: Initial Release* (Version v1.0.0). Zenodo. https://doi.org/10.5281/zenodo.10255963

Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, *34*(3), 596–615. https://doi.org/10.1145/28869.28874

Pulido, F. J., Mandow, L., & Pérez de la Cruz, J. L. (2014). Multiobjective shortest path problems with lexicographic goal-based preferences. *European Journal of Operational Research*, *239*(1), 89–101. https://doi.org/10.1016/j.ejor.2014.05.008

Skriver, A. J. V., & Andersen, K. A. (2000). A label correcting approach for solving bicriterion shortest-path problems. *Computers & Operations Research*, *27*(6), 507–524. https://doi.org/10.1016/s0305-0548(99)00037-4

Raith, A., Schmidt, M., Schöbel, A., & Thom, L. (2018). Extensions of labeling algorithms for multi-objective uncertain shortest path problems. *Networks*, *72*(1), 84–127. https://doi.org/10.1002/net.21815

Demetrescu, C., Goldberg, A., & Johnson, D. (2009). 9th DIMACS Implementation Challenge - Shortest Paths [Accessed: 2021-12-15. http://www.diag.uniroma1.it//~challenge9/].

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107. https://doi.org/10.1109/TSSC.1968.300136

Goldberg, A. V., Kaplan, H., & Werneck, R. F. (2006). Reach for A*: Shortest path algorithms with preprocessing. *The Shortest Path Problem*.

Stewart, B. S., & White, C. C. (1991). Multiobjective A*. *Journal of the ACM*, *38*(4), 775–814. https://doi.org/10.1145/115234.115368

Mandow, L., & Pérez de la Cruz, J. L. (2005). A New Approach to Multiobjective A* Search. *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, 218–223.

Mandow, L., & Pérez de la Cruz, J. L. (2010). Multiobjective A* search with consistent heuristics. *Journal of the ACM*, *57*(5), 1–25. https://doi.org/10.1145/1754399.1754400

Denardo, E. V., & Fox, B. L. (1979). Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, *27*(1), 161–186. https://doi.org/10.1287/opre.27.1.161

Maristany de las Casas, P. (2023b). *Targeted Multiobjective Dijkstra Algorithm: Initial Release* (Version v1.0.0). Zenodo. https://doi.org/10.5281/zenodo.7702018

Foschini, L., Hershberger, J., & Suri, S. (2012). On the Complexity of Time-Dependent Shortest Paths. *Algorithmica*, *68*(4), 1075–1097. https://doi.org/10.1007/s00453-012-9714-7

Disser, Y., Müller-Hannemann, M., & Schnee, M. (2008). Multi-criteria shortest paths in time-dependent train networks. *Exp. Algorithms Lecture Notes Comput. Sci*, *5038*, 347–361. https://doi.org/10.1007/978-3-540-68552-4_26

Kostreva, M. M., & Lancaster, L. (2002). Multiple objective path optimization for time dependent objective functions. In T. Trzaskalik & J. Michnik (Eds.), *Multiple objective and goal programming* (pp. 127–142). Physica-Verlag HD. https://doi.org/10.1007/978-3-7908-1812-3_10

Hamacher, H. W., Ruzika, S., & Tjandra, S. A. (2006). Algorithms for time-dependent bicriteria shortest path problems. *Discrete Optimization*, *3*(3), 238–254. https://doi.org/10.1016/j.disopt.2006.05.006

da Silva, J. M., Ramos, G. d. O., & Barbosa, J. L. V. (2023). Multi-Objective Decision-Making Meets Dynamic Shortest Path: Challenges and Prospects. *Algorithms*, *16*(3). https://doi.org/10.3390/a16030162

Orda, A., & Rom, R. (1990). Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, *37*(3), 607–625. https://doi.org/10.1145/79147.214078

Williamson, D. P., & Shmoys, D. B. (2009). *The Design of Approximation Algorithms*. Cambridge University Press. https://doi.org/10.1017/cbo9780511921735

Papadimitriou, C. H., & Yannakakis, M. (2000). On the approximability of trade-offs and optimal access of Web sources. *Proceedings 41st Annual Symposium on Foundations of Computer Science*. https://doi.org/10.1109/sfcs.2000.892068

Tsaggouris, G., & Zaroliagis, C. (2007). Multiobjective Optimization: Improved FPTAS for Shortest Paths and Non-Linear Objectives with Applications. *Theory of Computing Systems*, *45*(1), 162–186. https://doi.org/10.1007/s00224-007-9096-4

Warburton, A. (1986). Approximation of Pareto Optima in Multiple-Objective, Shortest-Path Problems. *Operations Research*, *35*(1), 70–79. Retrieved December 4, 2023, from http://www.jstor.org/stable/170911

Bökler, F., & Chimani, M. (2020). Approximating Multiobjective Shortest Path in Practice. In *2020 proceedings of the twenty-second workshop on algorithm engineering and experiments (ALENEX)* (pp. 120–133). Society for Industrial; Applied Mathematics. https://doi.org/10.1137/1.9781611976007.10

Audet, C., Bigeon, J., Cartier, D., Digabel, S. L., & Salomon, L. (2021). Performance indicators in multiobjective optimization. *European Journal of Operational Research*, *292*(2), 397–422. https://doi.org/10.1016/j.ejor.2020.11.016

Bazgan, C., Jamain, F., & Vanderpooten, D. (2017). Discrete representation of the non-dominated set for multi-objective optimization problems using kernels. *European Journal of Operational Research*, *260*(3), 814–827. https://doi.org/10.1016/j.ejor.2016.11.020

Roditty, L., & Zwick, U. (2012). Replacement Paths and k Simple Shortest Paths in Unweighted Directed Graphs. *ACM Trans. Algorithms*, *8*(4). https://doi.org/10.1145/2344422.2344423

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, *7*(1), 48–50.

Prim, R. C. (1957). Shortest connection networks and some generalizations. *The Bell System Technical Journal*, *36*(6), 1389–1401. https://doi.org/10.1002/j.1538-7305.1957.tb01515.x

Fernandes, I. F. C., Goldbarg, E. F. G., Maia, S. M. D. M., & Goldbarg, M. C. (2020). Empirical study of exact algorithms for the multi-objective spanning tree. *Computational Optimization and Applications*, *75*(2), 561–605. https://doi.org/10.1007/s10589-019-00154-1

Hamacher, H. W., & Ruhe, G. (1994). On spanning tree problems with multiple objectives. *Annals of Operations Research*, *52*(4), 209–230. https://doi.org/10.1007/bf02032304

Ramos, R. M., Alonso, S., Sicilia, J., & González, C. (1998). The problem of the optimal biobjective spanning tree. *European Journal of Operational Research*, *111*(3), 617–628. https://doi.org/10.1016/S0377-2217(97)00391-3

Steiner, S., & Radzik, T. (2008). Computing all efficient solutions of the biobjective minimum spanning tree problem [Part Special Issue: Applications of OR in Finance]. *Computers & Operations Research*, *35*(1), 198–211. https://doi.org/10.1016/j.cor.2006.02.023

Sourd, F., & Spanjaard, O. (2008). A Multiobjective Branch-and-Bound Framework: Application to the Biobjective Spanning Tree Problem. *INFORMS Journal on Computing*, *20*(3), 472–484. https://doi.org/10.1287/ijoc.1070.0260

Amorosi, L., & Puerto, J. (2022). Two-phase strategies for the bi-objective minimum spanning tree problem. *International Transactions in Operational Research*, *29*(6), 3435–3463. https://doi.org/10.1111/itor.13120

Ehrgott, M., & Gandibleux, X. (2000). A survey and annotated bibliography of multiobjective combinatorial optimization. *OR Spectrum*, *22*(4), 425–460. https://doi.org/10.1007/s002910000046

Di Puglia Pugliese, L., Guerriero, F., & Santos, J. L. (2014). Dynamic programming for spanning tree problems: Application to the multi-objective case. *Optimization Letters*, *9*(3), 437–450. https://doi.org/10.1007/s11590-014-0759-1

Santos, J. L., Di Puglia Pugliese, L., & Guerriero, F. (2018). A new approach for the multiobjective minimum spanning tree. *Computers And Operations Research*, *98*, 69–83. https://doi.org/https://doi.org/10.1016/j.cor.2018.05.007

Lacour, R. (2014). *Exact and approximate solving approaches in multi-objective combinatorial optimization, application to the minimum weight spanning tree problem* (Theses 2014PA090067). Université Paris Dauphine - Paris IX. https://theses.hal.science/tel-01134242

Corley, H. W. (1985). Efficient spanning trees. *Journal of Optimization Theory and Applications*, *45*(3), 481–485. https://doi.org/10.1007/bf00938448

Perny, P., & Spanjaard, O. (2005). A preference-based approach to spanning trees and shortest paths problems. *European Journal of Operational Research*, *162*(3), 584–601. https://doi.org/10.1016/j.ejor.2003.12.013

Cayley, A. (2009). A theorem on trees. In *The collected mathematical papers* (pp. 26–28). Cambridge University Press. https://doi.org/10.1017/CBO9780511703799.010

Chen, G., Chen, S., Guo, W., & Chen, H. (2007). The multi-criteria minimum spanning tree problem based genetic algorithm. *Information Sciences*, *177*(22), 5050–5063. https://doi.org/10.1016/j.ins.2007.06.005

Maristany de las Casas, P. (2023c). *IG-MDA and BN Algo - Release for Preprint Citation* (Version v1.0.0-beta). https://doi.org/10.5281/zenodo.7842666

Knowles, J., & Corne, D. (2001). A comparison of encodings and algorithms for multiobjective minimum spanning tree problems. *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, *1*, 544–551 vol. 1. https://doi.org/10.1109/CEC.2001.934439

Boost. (2022). Boost C++ Libraries. Retrieved February 15, 2023, from http://www.boost.org/

PassMark-Software. (2023). CPU Benchmarks [Accessed April 27, 2023]. https://www.cpubenchmark.net/compare/2448vs3521/Intel-i7-4720HQ-vs-Intel-Xeon-Gold-6246

Clarke, S., Krikorian, A., & Rausen, J. (1963). Computing the n best loopless paths in a network. *Journal of the Society for Industrial and Applied Mathematics*, *11*(4), 1096–1102. Retrieved August 8, 2023, from http://www.jstor.org/stable/2946497

Eppstein, D. (2016). K-Best Enumeration. In M.-Y. Kao (Ed.), *Encyclopedia of algorithms* (pp. 1003–1006). Springer New York. https://doi.org/10.1007/978-1-4939-2864-4_733

Lawler, E. L. (1972). A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, *18*(7), 401–405. Retrieved May 4, 2023, from http://www.jstor.org/stable/2629357

Yen, J. Y. (1972). Finding the Lengths of All Shortest paths in N-Node Nonnegative-Distance Complete Networks Using 1/2 N3 Additions and N3 Comparisons. *Journal of the ACM (JACM)*, *19*(3), 423–424.

Martins, E. Q. V., & Pascoal, M. M. B. (2003). A new implementation of Yen's ranking loopless paths algorithm. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, *1*(2). https://doi.org/10.1007/s10288-002-0010-2

Kurz, D., & Mutzel, P. (2016). A Sidetrack-Based Algorithm for Finding the k Shortest Simple Paths in a Directed Graph. https://doi.org/10.48550/ARXIV.1601.02867

Eppstein, D. (1998). Finding the k Shortest Paths. *SIAM Journal on Computing*, *28*(2), 652–673. https://doi.org/10.1137/s0097539795290477

Feng, G. (2014a). Finding k shortest simple paths in directed graphs: A node classification algorithm. *Networks*, *64*(1), 6–17. https://doi.org/10.1002/net.21552

Gotthilf, Z., & Lewenstein, M. (2009). Improved algorithms for the k simple shortest paths and the replacement paths problems. *Information Processing Letters*, *109*(7), 352–355. https://doi.org/10.1016/j.ipl.2008.12.015

Pettie, S. (2004). A new approach to all-pairs shortest paths on real-weighted graphs [Automata, Languages and Programming]. *Theoretical Computer Science*, *312*(1), 47–74. https://doi.org/https://doi.org/10.1016/S0304-3975(03)00402-X

Orlin, J. B., & Végh, L. (2022). Directed Shortest Paths via Approximate Cost Balancing. *J. ACM*, *70*(1). https://doi.org/10.1145/3565019

Sedeño-Noda, A. (2016). Ranking One Million Simple Paths in Road Networks. *Asia-Pacific Journal of Operational Research*, *33*(05), 1650042. https://doi.org/10.1142/s0217595916500421

Feng, G. (2014b). Improving Space Efficiency With Path Length Prediction for Finding k Shortest Simple Paths. *IEEE Transactions on Computers*, *63*(10), 2459–2472. https://doi.org/10.1109/TC.2013.136

Martins, E. d. Q. V., Pascoal, M. M. B., & Santos, J. L. (1999). Deviation Algorithms For Ranking Shortest Paths. *International Journal of Foundations of Computer Science*, *10*(03), 247–261. https://doi.org/10.1142/s0129054199000186

Roditty, L., & Zwick, U. (2005). Replacement Paths and k Simple Shortest Paths in Unweighted Directed Graphs. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, & M. Yung (Eds.), *Automata, languages and programming* (pp. 249–260). Springer Berlin Heidelberg.

Gorski, J., Klamroth, K., & Sudhoff, J. (2022). Biobjective optimization problems on matroids with binary costs. *Optimization*, *72*(7), 1931–1960. https://doi.org/10.1080/02331934.2022.2044479

Williams, V. V., & Williams, R. R. (2018). Subcubic Equivalences Between Path, Matrix, and Triangle Problems. *Journal of the ACM*, *65*(5), 1–38. https://doi.org/10.1145/3186893

Maristany de las Casas, P. (2023d). *maristanyPedro/kshortestpaths: Initial Release – Preprint Citation* (Version v1.0.0-beta). Zenodo. https://doi.org/10.5281/zenodo.8324671